X3T11/97-
X3T11/Project xxxx-D/REV 0.7

HIGH-PERFORMANCE PARALLEL INTERFACE - Scheduled Transfer

(HIPPI-ST)

June 5, 1997

Secretariat:

Information Technology Industry Council (ITI)

ABSTRACT: This standard specifies a data transfer protocol that uses small control messages to pre-arrange data movement. Buffers are allocated at each end before the data transmission, allowing full-rate, non-congesting data flow between the end devices. The control and data may use different physical media, or may share a single physical medium. Procedures are provided for moving data over HIPPI and other media.

NOTE:

This is an internal working document of X3T11, a Technical Committee of Accredited Standards Committee X3. As such, this is not a completed standard. The contents are actively being modified by X3T11. This document is made available for review and comment only. For current information on the status of this document contact the individuals shown below:

POINTS OF CONTACT:

Roger Cummings (X3T11 Chairman)
Distributed Processing Technology
140 Candace Drive
Maitland, FL 32751
(407) 830-5522 x348, Fax: (407) 260-5366
E-mail: cummings_roger@dpt.com

Ed Grivna (X3T11 Vice-Chairman)
Cypress Semiconductor
2401 East 86th Street
Bloomington, MN 55425
(612) 851-5200, Fax: (612) 851-5087
E-mail: elg@cypress.com

Don Tolmie (HIPPI-ST Technical Editor)
Los Alamos National Laboratory
CIC-5, MS-B255
Los Alamos, NM 87545
(505) 667-5502, Fax: (505) 665-7793
E-mail: det@lanl.gov

Comments on Rev 0.7

This is a preliminary document undergoing lots of changes. Many of the additions are just place holders, or are put there to stimulate discussion. Hence, do not assume that the items herein are correct, or final – everything is subject to change. This page tries to outline where we are; what has been discussed and semi-approved, and what has been added or changed recently and deserves your special attention. This summary relates to changes since the previous revision. Also, previous open issues are outlined with a single box, new open issues ones are marked with a double bar on the left edge of the box.

Changes are marked with margin bars so that changed paragraphs are easily found, and then highlights mark the specific changes. The list below just describes the major changes, for detail changes please compare this revision to the previous revision. **The major technical changes are printed in bold.**

Please help us in this development process by sending comments, corrections, and suggestions to the Technical Editor, Don Tolmie, of the Los Alamos National Laboratory, at det@lanl.gov. If you would like to address the whole group working on this document, send the comment(s) to hippi@network.com.

1. In the Foreword, deleted the bullet that said “This standard provides an upward growth path for legacy HIPPI-based systems”.
2. In the Introduction, deleted the bullet that said “Mappings from IPv4, IPv6, and MPI upper-layer protocols into Scheduled Transfers” since these will be documented in other places, e.g., RFCs.
3. **Deleted everything associated with Concatenate and Source_Concatenate. Since these were deletions, some of the highlights and margin bars are somewhat cryptic. This is considered a major technical change, but only this change item is listed in bold.**
4. In 3.1.10, changed “…carried separately…” to “…carried in the Schedule Header separately…”.
5. In 3.3, and throughout the document, changed "END" to "End", and "ACK" to "Ack".
6. In 4.2, split the paragraph into two paragraphs, and the figure into two figures, and did some minor rewording, all for improved clarity.
7. In figure 5, changed "Remote end" to "Originating Source", "Local end" to "Final Destination", "T_id$_n$" with "S_id$_n$" and "R_id$_n$", and "B_num$_n$" to "B_id$_n$".
8. **In 4.3.4, changed the maximum Buffer size from 2^{64} to 2^{63} bytes.** Changed the representation for Bufsize to "$2^{Bufsize}$ where $8 \leq Bufsize \leq 63$".
9. **In 4.3.5, changed an STU's maximum payload size from 2^{32} to 2^{31} bytes**. Changed the representation for Max-STU to "$2^{Max\text{-}STU}$ where $8 \leq Max\text{-}STU \leq 31$".
10. In 4.3.6, added that RQPs don't consume slots. Changed the last few sentences so that it is incumbent upon the Originating Source to keep a slot in reserve instead of having the Final Destination advertise one too few slots. In the last paragraph, added some text about knowing when to update Slots vision.
11. In 4.4.1, close to the end, changed "…are used…" to "…may be used…".
12. In 4.4.2, split the original last sentence of the first paragraph into two sentences so that the **exception of using B_id, instead of T_id, in Request_To_Receive and Data Operations can be stated in the last sentence.**
13. **In 4.4.3, added the whole new clause specifying the Block identifier (B_id). Previously all Blocks of a Transfer used the same identifier, and now each Block can have a different value.**
14. **In 4.4.6, changed the maximum Block size from 2^{64} to 2^{63} bytes.** Changed the representation for Blocksize to "$2^{Blocksize}$ where $8 \leq Blocksize \leq 63$".
15. In 4.4.8, deleted the concept of Bufx containing part of a concatenated address.
16. In 4.4.9, deleted the concept of OS_Bufx containing part of a concatenated address.
17. In 4.4.10, clarified where the Opaque data is carried. **Deleted the specification about**

which Schedule Header field contains the most and least significant parts of the Opaque data.

18. In 4.4.11, second paragraph, changed "No Offset is required…" to "Offset = zeros is required…".
19. In 5, changed "…a set of HIPPI-ST services must be supplied sufficient to", "…a sufficient set of HIPPI-ST services must be supplied to…".
20. **In figure 8, the T_len field was renamed "Sync".** This change ripples throughout the document. Note that it is just the field that changed name, the 64-bit T_len parameter did not change.
21. In 6.1, under Bufx, Offset, OS_Bufx, and OS_Offset, the text was changed to delete the concatenated address mode. **Under R_id, broke it into two subsections for carrying R_id and B_id**. **Under Sync, added the text for CTS Operations containing 0's in the high-order 16-bits and B_id in the low-order 16 bits.**
22. **In 6.2, under Silent, "OS_Bufx" was changed to "Bufx"**, and "semiotics" was changed to "semantics".
23. **In figure 8 and 6.2, the Concatenate and Source_Concatenate bits were changed to "Reserved".**
24. In 7, the use of the Concatenate and Source_Concatenate flags was deleted.
25. **In 7.1, the T_len field was changed to Sync,** and the use of the Concatenate and Source_Concatenate flags was deleted.
26. **In 7.2, the T_len field was changed to Sync,** and the use of the Concatenate and Source_Concatenate flags was deleted**.**.
27. In 7.3, added the last portion of the first paragraph reading "…that decreases timeout dependency for releasing resources.
28. **In 8.1, the T_len field was changed to Sync. Changed the Op code from x'06' to x'16'.** The last paragraph was reworded for clarity.
29. **In 8.2, changed the Op code from x'07' to x'17'.** The use of the Concatenate and Source_Concatenate flags was deleted.
30. In 8.3, the first paragraph was changed with the deletion of the Concatenate and Source_Concatenate flags. Added "…(the initiator)…" for clarity. The third paragraph had some rewording for clarity. **Changed the Op code from x'08' to x'18', and the T_len field was changed to Sync. Changed the R_id parameter to B_id.** Added text for B_id.
31. In 8.4, the first and last paragraphs were changed to indicate that an RTRR may be issued to indicate that the associated Data Operation will be delayed. **Changed the Op code from x'09' to x'19'**.
32. **In 8.5, changed the Op code from x'0A' to x'1A'.** The use of the Concatenate and Source_Concatenate flags was deleted. **Added the Block identifier (B_id) parameter** to the semantics list, and descriptive text below it.
33. **In 8.6, changed the Op code from x'0B' to x'1B'.** The use of the Concatenate and Source_Concatenate flags was deleted. **Changed the R_id parameter to B_id**, in the semantics and the text accordingly. Sync is now carried in the Sync field, so the semantics and the text under "Sync", were changed accordingly.
34. **In 8.7, changed the Op code from x'0C' to x'1C'.** The use of the Concatenate and Source_Concatenate flags was deleted. Sync is now carried in the Sync field, so the semantics and the text under "Sync", were changed accordingly.
35. **In 8.8, changed the Op code from x'0D' to x'1D'.** The use of the Reject, Concatenate and Source_Concatenate flags was deleted. Sync is now carried in the Sync field, so the semantics and the text under "Sync", were changed accordingly.
36. **In 8.9, changed the Op code from x'0E' to x'1E'.** The name were changed from END to End, and from END_ACK to End_Ack.
37. **In 8.10, changed the Op code from x'0F' to x'1F'.** The names were changed from END to End, and from END_ACK to End_Ack.
38. In table 2, the Concatenate and Source_Concatenate flags were deleted. The T_len field was renamed to Sync. The Sync value for PT, PTA and PTC were changed to *.

39. In table 3, the Concatenate and Source_Concatenate flags were deleted. The T_len field was renamed to Sync. The Op values were updated to match the semantics in 8.1 through 8.10, i.e., changed the high Op bit from 0 to 1. Changed R_id to B_id for RTR and Data. Added zeros and B_id to CTS in the Sync field. Changed to a single field for Opaque data in Data Operations.
40. In 9.3.1, deleted the text about undefined Opcode for "future supersets".
41. **In 9.4.4, changed the maximum sizes from 64 to 63, and from 2^{64} to 2^{63}.**
42. Between 9.5.2 and 9.5.3, deleted the clauses associated with Concatenate and Source_Concatenate.
43. **In 9.5.5, added the new clause for checking for Block out of order.**
44. **In 9.5.6, changed 64 to 63, and 2^{64} to 2^{63}.**
45. In 9.5.7, changed "…have not been received…" to "…have not occurred…". **Changed "…Request_State_Response…" to "…Request_To_Receive_Response…".**
46. In 9.5.8, changed "…Error logged." to "…Error should be logged."
47. In table 5, added the Out_Of_Order_B_num logged error. Fixed a few other typos.
48. In figures A.1 and A.2, changed the D_ULA and S_ULA fields to be contiguous (consistent with HIPPI-6400-PH) rather than show the horizontal bar between the 32 bit words. Changed the T_len field to Sync.
49. Annex B had lots of editorial changes to the text, but no substantive technical changes. For example, all of the Operations were spelled out, e.g., CTS to Clear_To_Send.
50. Annex C had lots of editorial changes, but we were not able to include all of the changes we would like in this revision. Hence, take it for what is there now and stay tuned for future revisions. None of the changes are marked with margin bars or highlights due to the extensive nature of the changes, i.e., you are better off to start reading fresh.

Contents

Page

Tables

Figures

Annexes

Foreword (This foreword is not part of American National Standard X3.xxx-199x.)

This American National Standard specifies a data transfer protocol that uses small control messages to pre-arrange data movement. Buffers are allocated at each end before the data transmission, allowing full-rate, non-congesting data flow between the end devices. The control and data may use different physical media, or may share a single physical medium. Procedures are provided for moving data over HIPPI and other media.

This document includes annexes which are informative and are not considered part of the standard.

Requests for interpretation, suggestions for improvement or addenda, or defect reports are welcome. They should be sent to the X3 Secretariat, Information Technology Industry Council, 1250 Eye Street, NW, Suite 200, Washington, DC 20005.

This standard was processed and approved for submittal to ANSI by Accredited Standards Committee on Information Processing Systems, X3. Committee approval of the standard does not necessarily imply that all committee members voted for approval. At the time it approved this standard, the X3 Committee had the following members:

> (List of X3 Committee members to be included in the published standard by the ANSI Editor.)

Subcommittee X3T11 on Device Level Interfaces, which developed this standard, had the following participants:

> (List of X3T11 Committee members, and other active participants, at the time the document is forwarded for public review, will be included by the Technical Editor.)

Introduction

This American National Standard specifies a data transfer protocol that uses small control messages to pre-arrange data movement. Buffers are allocated at each end before the data transmission, allowing full-rate, non-congesting data flow between the end devices. The control and data may use different physical media, or may share a single physical medium. Procedures are provided for moving data over HIPPI and other media.

Characteristics of a HIPPI-ST include:

- A hierarchy of data units (Scheduled Transfer Units (STUs), Blocks, and Transfers).
- Support for Get and Put Operations.
- Parameters exchanged between end devices for port selection, transfer identification, and Operation validation.
- Features supporting efficient mapping between the sender's and receiver's natural buffer sizes.
- Provisions for resending partial Transfers for error recovery.
- Mappings onto HIPPI-6400-PH, HIPPI-FP (for HIPPI-800 traffic), and Ethernet lower-layer protocols.

American National Standard for Information Technology –

High-Performance Parallel Interface – Scheduled Transfer (HIPPI-ST)

1 Scope

This American National Standard specifies a data transfer protocol that uses small control messages to pre-arrange data movement. Buffers are allocated at each end before the data transmission, allowing full-rate, non-congesting data flow between the end devices. The control and data may use different physical media, or may share a single physical medium. Procedures are provided for moving data over HIPPI and other media.

Specifications are included for:

- Virtual Connection setup and teardown;
- determining the number of Operations the other end can accept;
- determining the buffer size of the other end;
- exchanging Key, Port, transfer identifiers, and buffer size values specific to the end nodes;
- determining a maximum size transmission unit that will not overrun receiver buffer boundaries;
- using buffer indices and 64-bit addresses;
- acknowledging partial transfers so that buffers can be reused;
- providing means for resending partial Transfers for error recovery; and
- terminating transfers in progress.

Note that parts of the Scheduled Transfer protocol depend upon in-order delivery by the lower layer, which may not be available on all media.

2 Normative references

The following American National Standards contain provisions which, through reference in this text, constitute provisions of this American National Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below.

ANSI X3.183-1991, *High-Performance Parallel Interface – Mechanical, Electrical, and Signalling Protocol Specification (HIPPI-PH)*

ANSI X3.210-1992, *High-Performance Parallel Interface – Framing Protocol (HIPPI-FP)*

ANSI X3.xxx-199x, *High-Performance Parallel Interface – 6400 Mbit/s Physical Layer (HIPPI-6400-PH)*

ANSI/IEEE Std 802-1990, *IEEE Standards for Local and Metropolitan Area Networks: Overview and architecture (formerly known as IEEE Std 802.1A, Project 802: Local and Metropolitan Area Network Standard — Overview and Architecture).*

ISO/IEC 8802-2:1989 (ANSI/IEEE Std 802.2-1989), *Information Processing Systems – Local Area Networks – Part 2: Logical link control.*

3 Definitions and conventions

3.1 Definitions

For the purposes of this standard, the following definitions apply.

3.1.1 Block: An ordered set of one or more STUs within a Scheduled Transfer. (See figure 3 and 4.4.5.)

3.1.2 Buffer Index (Bufx): A 32-bit parameter identifying the starting address of a data buffer. Bufx may be either a pointer to the starting address or the most significant part of a 64-bit starting address.

3.1.3 Control Channel: The logical channel that carries the Control Operations.

3.1.4 Control Operation: A control function consisting of a Schedule Header and an optional 32-byte payload. (See figure 3.)

3.1.5 Data Channel: The logical channel that carries the data payload.

3.1.6 Data Operation: A data movement Operation consisting of a Schedule Header and up to 2 gigabytes of user payload. (See figure 3).

3.1.7 Final Destination: The end device that receives, and operates on, the data payload. This is typically a host computer system, but may also be a non-transparent translator, bridge, or router.

3.1.8 Key: A local identifier used to validate Operations. (See 4.3.3.)

3.1.9 log: The act of making a record of an event for later use.

3.1.10 Opaque data: Eight bytes of Source ULP to Destination ULP peer-to-peer information carried in the Scheduled Header separately from the data payload. (See 4.4.10)

3.1.11 Operation: A Scheduled Transfer function, i.e., a Control Operation or the data movement specified in an STU.

3.1.12 optional: Characteristics that are not required by HIPPI-ST. However, if any optional characteristic is implemented, it shall be implemented as defined in HIPPI-ST.

3.1.13 Originating Source: The end device that generates the data payload. This is typically a host computer system, but may also be a non-transparent translator, bridge, or router.

3.1.14 Persistent: A control mode used to retain buffers for multiple Transfers. (See 4.3.7.)

3.1.15 Port: A logical connection within an end device. (See 4.3.2.)

3.1.16 Scheduled Transfer: An information transfer, normally used for bulk data movement and low processing overhead, where the Originating Source and Final Destination prearrange the transfer using the protocol defined in this standard.

3.1.17 Scheduled Transfer Unit (STU): The data payload portion of a Data Operation moved from an Originating Source to a Final Destination. STUs are the basic components of Blocks. (See figure 3 and 4.4.7.)

3.1.18 Slot: A space reserved for a Control Operation, or the Schedule Header portion of an STU, in the end device. (See 4.3.6.)

3.1.19 Transfer: An ordered set of one or more Blocks within a Scheduled Transfer. (See figure 3 and 4.2.)

3.1.20 upper-layer protocol (ULP): The protocol above the service interface. These could be implemented in hardware, software, or they could be distributed between the two.

3.1.21 Virtual Connection: A bi-directional logical connection used for Scheduled Transfers between two end devices. A Virtual Connection contains a logical Control Channel and a logical Data Channel in each direction.

3.2 Editorial conventions

In this standard, certain terms that are proper names of signals or similar terms are printed in uppercase to avoid possible confusion with other uses of the same words (e.g., STU). Any lowercase uses of these words have the normal technical English meaning.

A number of conditions, sequence parameters, events, states, or similar terms are printed with the first letter of each word in uppercase and the rest lowercase (e.g., Block, Transfer). Any lowercase uses of these words have the normal technical English meaning.

The word *shall,* when used in this American National standard, states a mandatory rule or requirement. The word *should,* when used in this standard, states a recommendation.

3.2.1 Binary notation

Binary notation is used to represent relatively short fields. For example a two-bit field containing the binary value of 10 is shown in binary format as b'10'.

3.2.2 Hexadecimal notation

Hexadecimal notation is used to represent some fields. For example a two-byte field containing a binary value of b'11000100 00000011' is shown in hexadecimal format as x'C403'.

3.3 Acronyms and other abbreviations

Ack	acknowledge indication
CTS	Clear_To_Send
EndA	End_Ack
HIPPI	High-Performance Parallel Interface
K	kilo (2^{10} or 1024)
KB	kilobyte (1024 bytes)
MAC	Media Access Control
PT	Port_Teardown
PTA	Port_Teardown_Ack
PTC	Port_Teardown_Complete
RQP	Request_Port
RQPR	Request_Port_Response
RS	Request_State
RSR	Request_State_Response
RTR	Request_To_Receive
RTRR	Request_To_Receive_Response
RTS	Request_To_Send
RTSR	Request_To_Send_Response
STU	Scheduled Transfer Unit
ULP	upper-layer protocol

4 System overview

This clause provides an overview of the structure, concepts, and mechanisms used in Scheduled Transfers. Figure 1 gives an example of Scheduled Transfers being used to communicate between device A and device B over some physical media. Annex C describes the steps in a typical Scheduled Transfer. Figure 2 shows HIPPI-ST being used over different media.

4.1 Control Channels and Data Channels

Each Transfer has an Originating Source and Final Destination. Each Originating Source and Final Destination shall have a Control Channel and one or more Data Channels. The Originating Source sends the payload data, and the Final Destination receives the payload data.

Control Operations shall be exchanged over the Control Channel. Scheduled Transfer Units (STUs), i.e., data payload, shall be exchanged over the Data Channel(s). The information volume on the Data Channel(s) will be probably many times the volume on the Control Channel; hence the available bandwidths should be balanced accordingly. For best performance, the Control Channel should have low latency.

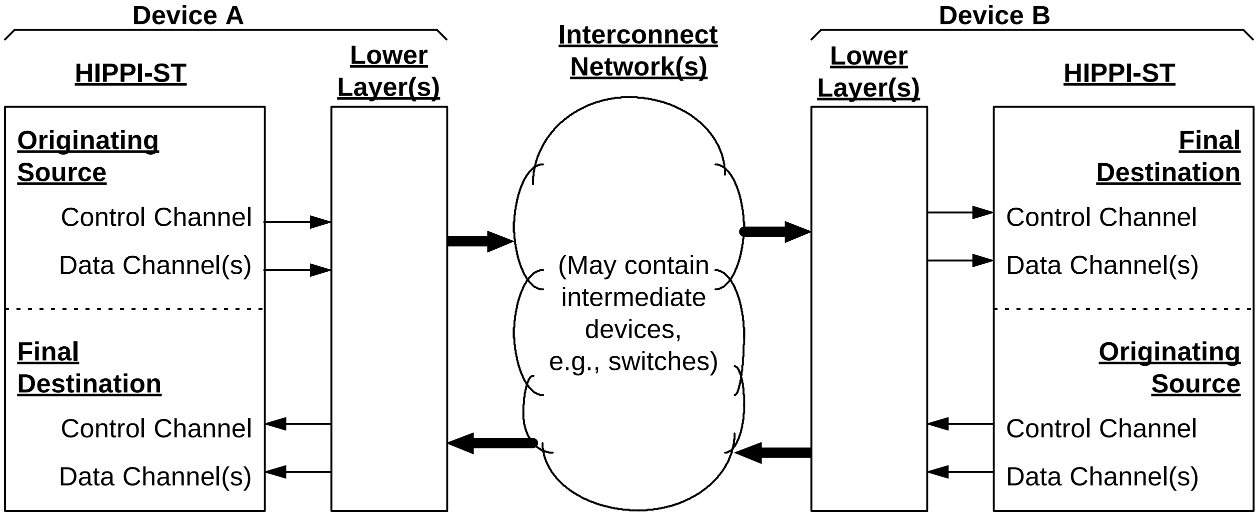


Figure 1 – System overview

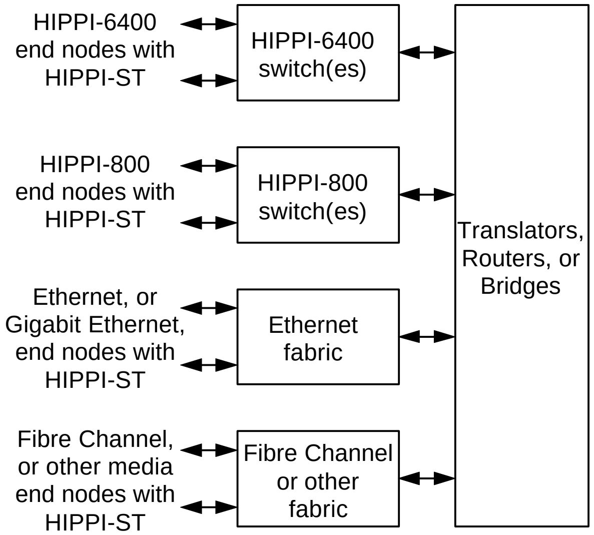

HIPPI-6400 end nodes with HIPPI-ST
HIPPI-6400 switch(es)
HIPPI-800 end nodes with HIPPI-ST
HIPPI-800 switch(es)
Ethernet, or Gigabit Ethernet, end nodes with HIPPI-ST
Ethernet fabric
Fibre Channel, or other media end nodes with HIPPI-ST
Fibre Channel or other fabric
Translators, Routers, or Bridges

Figure 2 – HIPPI-ST over different media

4.2 System model

Multiple write (Put) or read (Get) functions may be executed to move user data units, called Transfers, over a Virtual Connection. As shown in figure 3, a Transfer is composed of one or more Blocks, and Blocks are composed of one or more STUs. The Scheduled Transfer protocol shall package the Transfer in Blocks and STUs for delivery using lower layer protocol(s) and media.

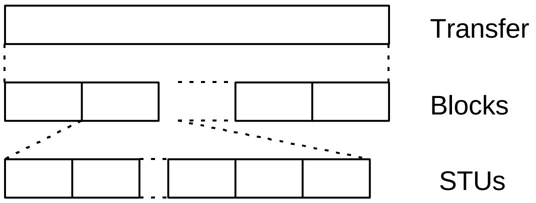


Figure 3 – User data hierarchy

As shown in figure 4, an STU shall be the data payload portion of a Data Operation. A Data Operation shall consist of a 40-byte Schedule Header and an STU of up to 2 gigabytes (2^{31} bytes). A Control Operation shall consist of a 40-byte Schedule Header, and may contain an additional 32 bytes of optional payload.

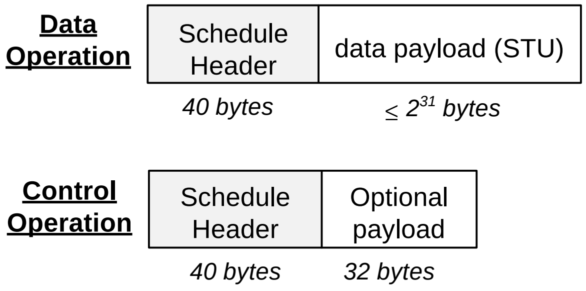


Figure 4 – Transmission units

Figure 5 shows the model used on a Final Destination for the Scheduled Transfers. The model on an Originating Source would be similar.

As Control Operations and Data Operations are received, the Schedule Header of each is placed in the Schedule Header queue for execution. State information about the number of empty Slots in the queue is available to the other end so that it can avoid overrunning the queue.

The Virtual Connection Descriptor contains:

– static parameters defining the Virtual Connection from the view of both the remote end device and local end device (the top portion of the Virtual Connection Descriptor box in figure 5);

– current state information about the number of empty "Slots" for Operation Schedule Headers, and Operation Retry and Timeout parameters;

– identifiers for each of the Virtual Connection's Transfers.

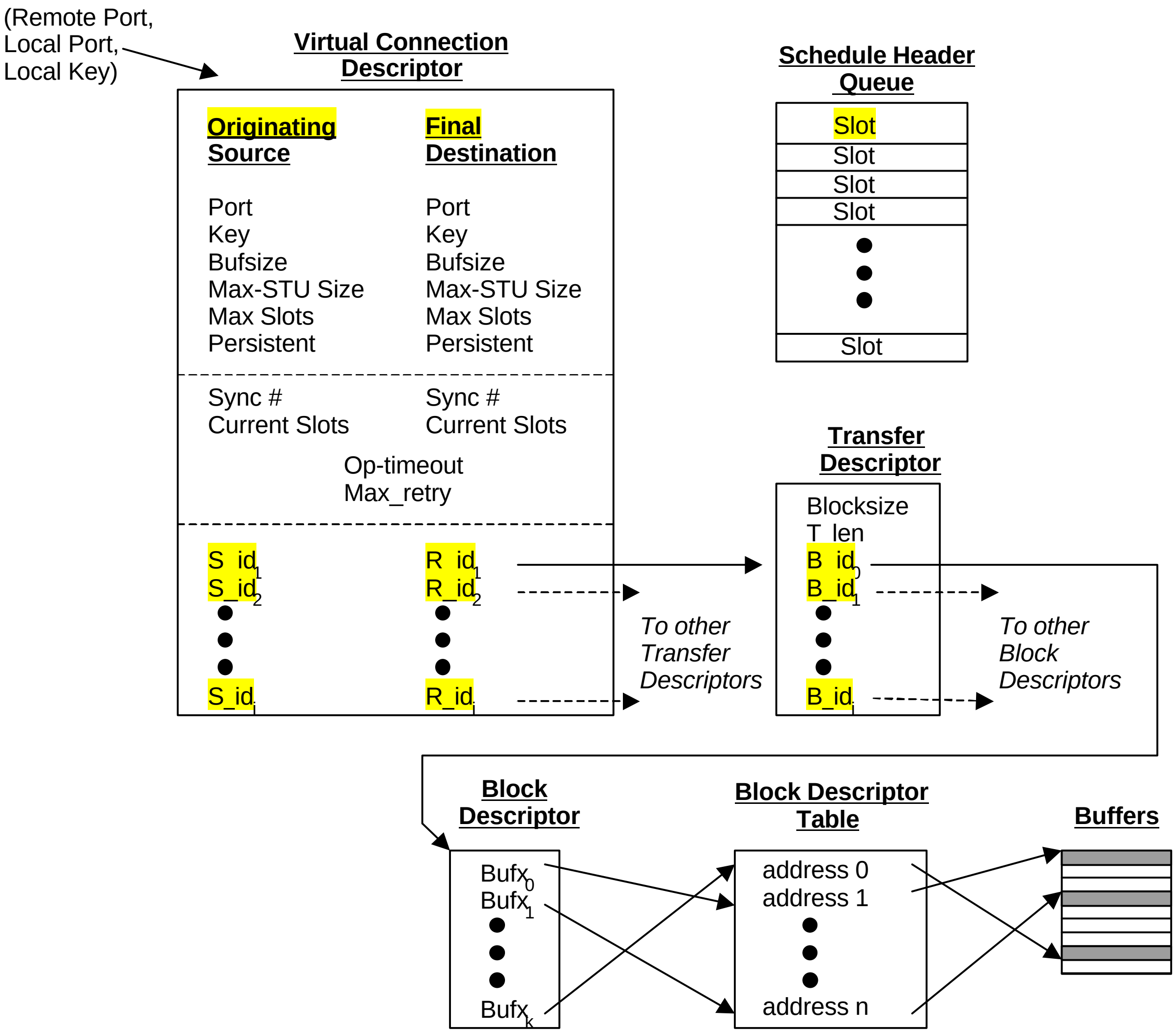


Figure 5 – Scheduled Transfer Final Destination model

A Transfer Descriptor, for each Transfer, contains the Transfer length (T_len, in bytes), the Block size (in bytes), and includes pointers to Block Descriptors. The Block Descriptors (one for each Block of a Transfer) identify the set of contiguous Buffer Index (Bufx) values assigned to the Block. And finally, the Buffer Descriptor Table provides a base memory address for each Bufx.

In an effort to achieve maximum transfer rates and efficiency, the receiver's job is made as easy as possible, even at the expense of the transmit side. It is expected that after validating an Operation in the Final Destination, only a single lookup will be needed to derive the absolute memory address and correctly place the data.

4.3 Virtual Connections

Scheduled Transfers between an Originating Source and Final Destination are pre-arranged to decrease computational overhead during the Transfer by allocating buffers at each end device. The bi-directional path between the end devices is called a Virtual Connection. A Virtual Connection shall consist of an Originating Source and Final Destination in each end device.

Once the Final Destination has indicated its ability to accept the STUs, the Virtual Connection should not become congested. In essence, the Final Destination smoothly controls the flow. For comparison, without pre-arranging the buffers, the Originating Source would blindly send data into the interconnection network where it might have to wait for buffers to be assigned in the Final Destination. On the down-side, Scheduled Transfers require additional Control Operations and round-trip latency. Once established, a Virtual Connection may be used to carry multiple Transfers. This Scheduled Transfer protocol does not handle network resource reservations.

4.3.1 Sequences and Operations

During Virtual Connection setup, the end devices shall exchange parameters specific to each device. These parameters, shown in the upper portion of the Virtual Connection Descriptor box in figure 5 and detailed below, include values for:

– Port numbers (e.g., a Port dedicated to HIPPI-FP or IP traffic);

– Keys (used for authenticating Operations);

– native buffer sizes (Bufsize) for determining Final Destination buffer tiling;

– maximum STU size;

– maximum number of outstanding Operations (Slots) to keep from overflowing the command queues;

– whether or not they support Persistent mode.

The parameters assigned during setup shall apply for the life of the Virtual Connection. Once established, the Virtual Connection is accessed as shown in figure 5 by the tuple "remote Port", "local Port", and "local Key". The Control Operations defined for Virtual Connection setup are:

– Request_Port (See 7.1.)

– Request_Port_Response (See 7.2.)

The Control Operations defined for Virtual Connection teardown are:

– Port_Teardown (See 7.3.)

– Port_Teardown_Ack (See 7.4.)

– Port_Teardown_Complete (See 7.5.)

4.3.2 Ports

Ports identify upper-layer entities within an end device. The Port values shall be assigned by the local end device and have no meaning on the other end device. For example, when end device A requests a Virtual Connection to end device B, A shall select the value for A-Port and shall send it to B in the Request_Port Operation. B shall store the A-Port value and shall return it to A in every Operation over this Virtual Connection. Likewise, B shall select the value for B-Port.

An exception is the "well-known Port", i.e., Port x'0000'. In this case, a request sent to the "well-known Port" shall result in the receiving end device assigning a specific local Port value based on the EtherType parameter. EtherType parameter values shall be as assigned in the current "Assigned Numbers" RFC, e.g., RFC 1700[1)]. For example, if the HIPPI-ST is used to encapsulate TCP/IP, then the EtherType would be x'0800'. If HIPPI-ST is being used to encapsulate legacy HIPPI-FP or user data, then the EtherTypes would be x'8180' and x'8181' respectively.

If the incoming Port number is invalid, then the Operations shall not be executed (see 9.4.1). A Port value of x'0000' is valid in Request_Port Operations; invalid in all other Operations.

4.3.3 Keys

Like the Ports, each end device shall select its own 32-bit Key value for use on the Virtual Connection. For example, when end device A requests a Virtual Connection to end device B, A shall select the value for A-Key and shall send it to B in the Request_Port Operation. B shall store the A-Key value and shall return it to A in every Operation over this Virtual Connection. The A-Key value has no meaning in B; it is only significant in A where it shall be used to validate that the Operation presented is really associated with this Virtual Connection. Likewise, B shall select the value for B-Key. Keys are similar in nature to passwords; if the Key doesn't match, then the Operation shall not be executed (see 9.4.1).

4.3.4 Buffer size (Bufsize)

Each end shall define the buffer size, in bytes, that it wants to use. Buffer sizes may be the same as host page sizes. It is most efficient when the buffer sizes are the same on both ends, but differing buffer sizes are supported (see annex C). The buffer sizes shall be ≥ 256 bytes and shall be an integral power of two, i.e., $2^{Bufsize}$ where $8 \leq Bufsize \leq 63$.

4.3.5 Max-STU size

The Max-STU size, exchanged during Virtual Connection setup, establishes the maximum data payload size of an STU (see 4.4.11). Each end device declares the desired Max-STU size it is prepared to receive. The Max-STU size must be no larger than its Bufsize. Intermediate devices with smaller buffer sizes may lower this value. Note that the Max-STU size in each direction may be different.

Additionally, an STU's maximum data payload size shall be ≥ 256 bytes and an integral power of two i.e., $2^{Max\text{-}STU}$ where $8 \leq Max\text{-}STU \leq 31$.

4.3.6 Slots and Sync parameter

The term Slot denotes memory at an end device reserved for storing the Schedule Header of an incoming Operation. Each Operation arriving at an end device consumes one Slot, except for Request_Port Operations, or for Data Operations which consume a Slot only if Silent = 0 or Interrupt = 1. An Originating Source shall control the flow of Operations by sending no more Operations than there are Slots available at the other end. Any Operations that are sent in excess of the number of available Slots may be discarded by the receiver (see 9.4.2). In order to avoid potential deadlocks that can happen if an Originating Source consumes all of its allocated slots at the Final Destination, an Originating Source shall never consume all of its slots with data movement Operations. Instead, and Originating Source shall hold at least one slot in reserve for possible use for an End, Request_State, Request_State_Response, or Port_Teardown sequence.

An end device learns the initial number of Slots available (Slots value) at the remote end device during the Virtual Connection setup (see 7.1 and 7.2). Later, an end device obtains the current Slots value by reading the Slots parameter in a received Request_State_Response. An end device may solicit a Request_State_Response from the remote end by either of two methods: by setting the Send_State flag in the Schedule Header of a Data Operation, or by sending a Request_State Operation. A received Slots value of x'FFFFFFFF' indicates that the remote end does not implement Slot accounting.

> NOTE – Slot flow control may not be needed when the maximum number of Control Operations is otherwise bounded or where dropped Operations are acceptable.

[1] RFC (Request For Comment) documents are working standards documents from the TCP/IP internetworking community. Copies of these documents are available from numerous electronic sources (e.g., http://www.ietf.org) or by writing to IETF Secretariat, c/o Corporation for National Research Initiatives, 1895 Preston White Drive, Suite 100 Reston, VA 20191-5434, USA.

The received Slot value is a snapshot of the number of Slots available at the remote end device when the remote end device received the soliciting Operation. The local end device may continue to send Operations after soliciting a Request_State_Response and may also solicit multiple responses before receiving a reply. The lower bound on the number of available Slots at the remote end device is determined by the local end device which adjusts its vision of the number of Slots to account for outstanding Operations. The adjustment consists of subtracting, from the number of Slots indicated in the received Request_State_Response Operation, the number of Slot-consuming Operations sent by the local end device after a Request_State_Response solicitation.

The local end device can use the Sync parameter to identify Request_State_Response messages when there are multiple outstanding solicitations, i.e., to know when to update, or not update, the number of available Slots at the Final Destination. The Sync parameter in a Data or Request_State shall be copied and returned by the remote end device in the corresponding Request_State_Response. The Sync parameter may be used by the local end device to mark the request, and thus identify the Request_State_Response with a particular solicitation. The Sync values are locally determined.

4.3.7 Persistent

The Persistent flag (see 6.2) controls buffer retention in the Final Destination for the Virtual Connection.

- When Persistent = 1, the memory in the Final Destination allocated for the Scheduled Transfer shall be retained for multiple transfers and not released until a Port_Teardown or an End Operation occurs. Note that Persistent = 1 bypasses the flow control provided by Clear_To_Send, i.e., a Data Operation may be sent at any time whether or not a Clear_To_Send Operation has been received. Sending information to a Frame Buffer is an example of where Persistent might be used.

- When Persistent = 0, the memory for a Block may be allocated for other uses after the Block is complete. All Data Operations must be enabled by a Clear_To_Send or Request_To_Receive Operation.

Persistent is only usable between hosts that mutually agree. Agreement is reached by controlling the Persistent flag bit during the Virtual Connection setup (see 7.1 and 7.2).

4.4 Data movement

4.4.1 Sequences and Operations

A write data sequence (which may be initiated by either end of the Virtual Connection) shall be set up by the end devices exchanging transfer identifiers (T_id's), specific to each device, and length parameters. The Control Operations setting up a write data sequence are:

– Request_To_Send (See 8.1.)

– RTS_Response (See 8.2.)

A read data sequence, which moves the Transfer as a single Block, requires that both ends had previously allocated resources for the entire read sequence with a Request_To_Send. The Control Operations setting up a read data sequence are:

– Request_To_Receive (See 8.3.)

– Request_To_Receive_Response (See 8.4.)

The Final Destination controls the data flow with:

– Clear_To_Send (See 8.5.)

Data payloads for the read and write data movements are carried in STUs. STUs are sent with:

– Data (See 8.6.)

State information can be requested in a Data Operation or with a Request_State Control Operation.

– Request_State (See 8.7.)

– Request_State_Response (See 8.8.)

The Control Operations below may be used to abort limited size Transfers. Unlimited size Transfers shall use this method to signal the end of the Transfer.

– End (See 8.9.)

– End_Ack (See 8.10.)

4.4.2 Transfer identifiers (R_id and S_id)

Like the Ports and Keys, each end device shall also select its own non-zero 16-bit Transfer identifier (T_id) value for a data movement on the Virtual Connection. For example, when end device S requests to write to end device R, S shall select the value for its T_id and shall send it to R in the Request_To_Send Operation. R shall store S's T_id value and shall return it to S in most Operations concerning this Transfer. Likewise, R shall select its T_id value and send it to S in a Request_To_Send_Response or Clear_To_Send Operation. For each Operation, the sender shall put its T_id in the S_id field. The sender shall put the receiver's T_id value in the R_id field for all except Request_To_Receive and Data Operations, in which case it shall put the B_id value (see 4.4.3) in the R_id field.

NOTE – The Virtual Connection is symmetrical; either end device may initiate a data movement. For example, S could be end device A that initiated the Virtual Connection setup, or it could be end device B. Different names were used for clarity.

4.4.3 Block identifier (B_id)

Each Block of a Transfer may use a different 16-bit Block identifier (B_id). B_id values shall be selected by the Final Destination and passed to the Originating Source in Request_To_Receive and Clear_To_Send Operations. The associated Data Operations shall echo the B_id value. Note that the B_id parameter is used instead of the R_id parameter in Request_To_Receive and Data Operations.

4.4.4 Transfer length (T_len)

The 64-bit Transfer length parameter (T_len) specifies the total number of data payload bytes in the Transfer. T_len does not include the Schedule Header or any lower-layer headers. T_len = all zeros shall indicate an unlimited size Transfer. An unlimited size Transfer is terminated by an End Operation (see 8.9).

4.4.5 Blocks

Scheduled Transfer flow control, striping, acknowledgments, and resource allocation are all done on a Block basis. Block numbers (B_num) shall be numbered starting at zero and shall increment by one for each following Block.

Blocks comprising a Transfer shall be enabled for transmission in sequential order unless both the Originating Source and Final Destination indicated Out_of_Order capability during the Virtual Connection setup. Note that Out_of_Order is necessary for selective retransmission to correct flawed Blocks, otherwise go-back-N retransmission must be used.

Request_State_Response Operations indicate the highest numbered Block received correctly by the Final Destination. Request_State_Response Operations can be requested by setting the Send_State flag bit in Data Operations or by sending Request_State Operations. In addition, Request_State Operations can ask if a particular Block was received correctly. Use of these mechanisms allows the Originating Source to verify correct reception and to identify flawed Blocks for potential retransmission.

4.4.6 Block size

The Block size (the number of bytes in a Block) for a Transfer is established when a Transfer is initiated, i.e., with a Request_To_Send_Response or Clear_To_Send Operation (see 8.2 and 8.5). The Blocksize parameter is expressed as a power of two, i.e., $2^{\text{Blocksize}}$ where $8 \leq \text{Blocksize} \leq 63$. All of the Blocks of a Transfer shall be full size, except for the first and/or last Block of a Transfer which can be smaller (the first Block will be smaller by the initial Offset value, and the last Block will be whatever completes the Transfer).

4.4.7 STUs

The STUs of a Block shall be transmitted in order. STU numbers (S_count) shall start with zero and increment by one for each following STU. The last STU of a Block shall be marked with Last = 1. No STU shall extend past a Final

Destination's buffer boundary, Blocksize boundary, or Transfer boundary.

4.4.8 Bufx and Offset

Bufx contains a Buffer Index. If more than one Buffer Index is required for a Block, i.e., buffer size (Bufsize) is less than Blocksize, then the Bufx parameter in the Clear_To_Send Operation shall specify the initial Bufx, and any additional Bufx values shall be sequential.

Offset may be used to start at other than the first byte of a Final Destination's buffer. For the first STU of a Block, the Offset value shall be the same as received in the Clear_To_Send for the Block. Subsequent STUs of the Block shall adjust the Bufx and Offset values based on the Final Destination's buffer size and the STU size used by the Originating Source.

The Offset value associated with the first block of a Transfer (I_Offset) is included in all Clear_To_Send Operations. This allows the Originating Source to compute the starting address for any Block without having received the Clear_To_Send for the first Block. Clear_To_Send Operations can occur out of order, e.g., as the result of striping.

Best performance will usually be achieved when an Offset value of zero is specified. Use of non-zero offset values may degrade performance, depending upon underlying hardware transfer mechanisms.

4.4.9 OS_Bufx and OS_Offset

OS_Bufx specifies a Buffer Index. If more than one Buffer Index is required for a Block, i.e., buffer size < Block size, then the OS_Bufx parameter shall specify the initial Bufx, and any additional Bufx values shall be sequential.

OS_Offset may be used to start at other than the first byte of a Source buffer. Note that OS_Bufx and OS_Offset are only used with Request_To_Receive Operations, and Request _To_Receive Operations only specify one Block.

4.4.10 Opaque data

Opaque data is eight bytes of ULP peer-to-peer information carried in a Data Operation’s Schedule Header OS_Bufx and OS_Offset fields. The Opaque data shall be delivered to the Final Destination’s ULP when Silent = 0 (see 6.2). The Opaque data shall be passed unmodified from the Originating Source to the Final Destination. Note that the Opaque data uses Slot resources while the data payload uses Bufx resources. The Opaque data shall not be counted in the length, tiling, or Bufx calculations.

4.4.11 Packing examples

Figure 6 shows three possibilities for packing the same Transfer into a receiver's buffers. All three examples show a group of seven of the receiver's buffers on the top line. Each buffer is pointed to by a Bufx, and the data in the first buffer starts at an Offset value. The Transfer is the shaded bar, with transmission going from left to right. The Block boundaries are shown above the shaded bar, and the resulting STU boundaries are shown below the shaded bar.

Example (a), at the top, shows the case where the buffers and Blocks are the same size. Notice that the first Block is smaller than the other Blocks by the Offset value. Offset = zeros is required for the other Blocks. The last Block of the Transfer is also smaller, i.e., the Transfer did not end on a Block boundary. While the STU boundaries lined up nicely, the sender could have used multiple STUs, but the STUs cannot be larger than Max-STU.

Example (b) shows multiple Blocks per receiver buffer. The Blocks that do not start on a buffer boundary would use the Offset parameter to position the data.

Example (c) shows the Block size covering two of the receiver's buffers.

In summary, STUs cannot cross Block, buffer, or Transfer boundaries. Relationships include:

STU size ≤ Max-STU size

Max-STU size ≤ Blocksize

Max-STU size ≤ Bufsize

Note that the Blocksize can be larger, smaller, or the same as Bufsize.

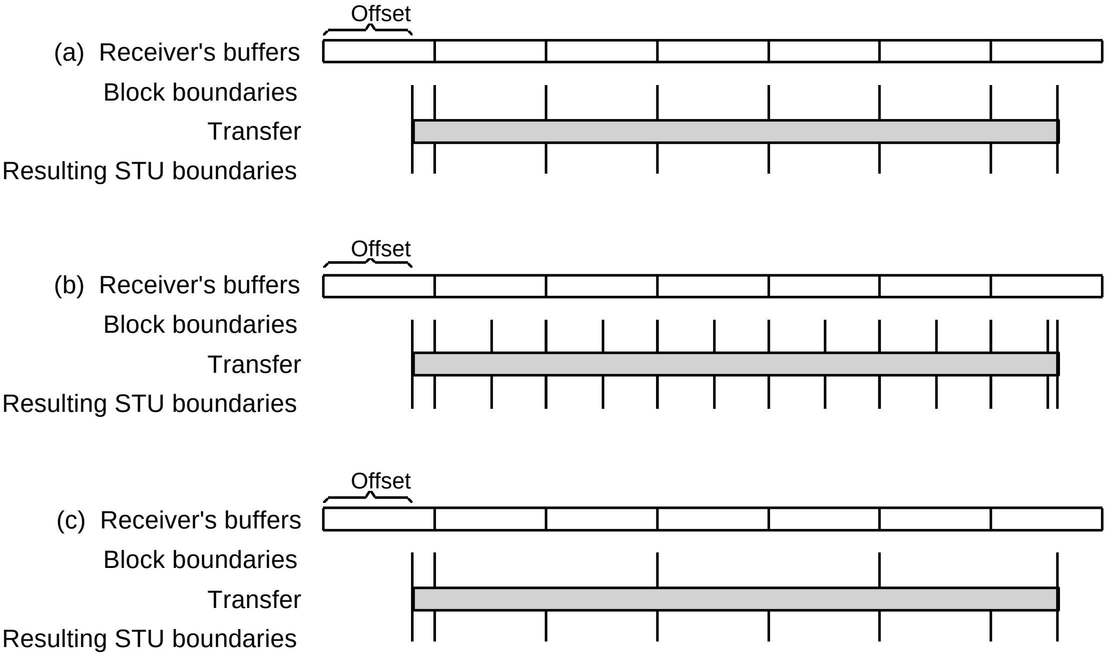


Figure 6 – Data packing examples

4.5 Operations management

4.5.1 Flow control

Data flow control is achieved with Clear_To_Send and Request_To_Receive Operations; each one sent by the Final Destination gives the Originating Source permission to send one Block. Flow control is overridden when Persistent = 1; here Data Operations may be sent without having first received Clear_To_Send Operations.

Operation flow control is achieved by an Operation's sender not overrunning the Slots value (see 4.3.6).

4.5.2 Status Operations

Request_State (see 8.7) and Request_State_Response (see 8.8) Operations are used to request and supply status information about the state of the remote end device. They can be used to see which Blocks have been received correctly and the number of empty Slots available. The Sync parameter (see 4.3.6) is used to provide a common reference point for the local and remote end devices, i.e., to match Request_State and Request_State_Response Operations.

4.5.3 Rejected Operations

If the receiving end device is unable to execute an Operation, then the receiving device shall set the Reject flag bit = 1 in the response. Table 1 shows the response when an Operation is rejected. The recovery actions taken when an Operation is rejected are beyond the scope of this standard.

Table 1 – Response to a rejected Operation

Rejected Operation	**Response** (w/ Reject = 1)
Request_Port	Request_Port_Response
Request_To_Send	Request_To_Send_Response
Request_To_Receive	Request_To_Receive_Response

4.5.4 Lost Operations

Errors other than syntactic errors are manifested as missing Operations, which occur when the underlying physical medium discards or damages a transmission. Each Scheduled Transfer Operation is defined as part of a two-way handshake or a three-way handshake. Thus, for each command Operation there is a corresponding response Operation, and for some response Operations there is also a corresponding completion Operation.

Each Operation that expects a response is guarded with a timeout whose value is referred to as Op_timeout (see 9.1). An Operation shall be re-tried up to Max_Retry times (see 9.1) if the sending end device does not receive the expected response (see 9.2 and table 5).

Data transmissions (i.e., Data Operations) are an exception to this timeout mechanism and are referred to the ULP for resolution (see 9.2).

4.5.5 Interrupts

An Interrupt causes a signal to be delivered to the receiving end device ULP. An Interrupt can be requested with any Operation by setting Interrupt = 1.

5 Service interface

This clause specifies the services provided by HIPPI-ST. The intent is to allow ULPs to operate correctly with this HIPPI-ST. How many of the services described herein are chosen for a given implementation is up to that implementor; however, a sufficient set of HIPPI-ST services must be supplied to satisfy the ULP(s) being used. The services as defined herein do not imply any particular implementation or any interface.

Figure 7 shows the relationship of the HIPPI-ST interfaces.

5.1 Service primitives

The primitives, in the context of the state transitions in clause 5, are declared required or optional. Additionally, parameters are either required, conditional, or optional. All of the primitives and parameters are considered as required except where explicitly stated otherwise.

HIPPI-ST service primitives are of four types.

– *Request primitives* are issued by a service user to initiate a service provided by the HIPPI-ST. In this standard, a second Request primitive of the same name shall not be issued until the Confirm for the first request is received.

–*Confirm primitives* are issued by the HIPPI-ST to acknowledge a Request.

– *Indicate primitives* are issued by the HIPPI-ST to notify the service user of a local event. This primitive is similar in nature to an unsolicited interrupt. Note that the local event may have been caused by a service Request. In this standard, a second Indicate primitive of the same name shall not be issued until the Response for the first Indicate is received.

– *Response primitives* are issued by a service user to acknowledge an Indicate.

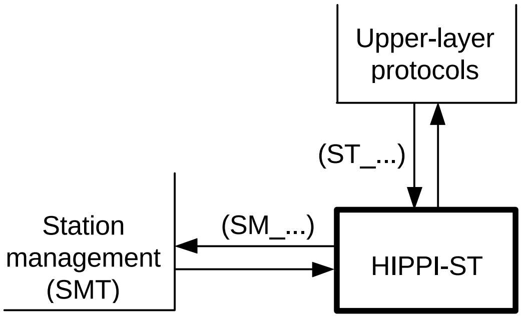


Figure 7 – HIPPI-ST service interface

5.2 Sequences of primitives

The order of execution of service primitives is not arbitrary. Logical and time sequence relationships exist for all described service primitives. Time sequence diagrams are used to illustrate a valid sequence. Other valid sequences may exist. The sequence of events between peer users across the user/provider interface is illustrated. In the time sequence diagrams, the HIPPI-ST users are depicted on either side of the vertical bars, while the HIPPI-ST acts as the service provider.

NOTE - The intent is to flesh out the service primitives similar to what is in HIPPI-PH today.

Service interface considerations -

(These are notes that have been collected during the document reviews, and should be considered when the service interface is written.)

Should there be a priority, or time-to-live, for individual Transfers? On a per connection basis?

Pass the full ST header to/from the ULP.

Service the slots in order of arrival, i.e., FIFO.

Interrupts are passed independent of the Slots, i.e., whenever an Interrupt is put in the Slots queue.

There is a Port-basis for the Service Interface

6 Schedule Header

The Schedule Header is shown in figure 8 as a group of 32-bit words. The Schedule Header fields are named for the most common parameter for which the field is used. Many of the fields have different uses depending on the Operation type, and some Operations do not use one or more of the fields at all. The usage for each field is listed below and summarized in tables 2 and 3.

			Bytes
Op	Flags	S_count	00-03
R_Port		S_Port	04-07
Key			08-11
R_id		S_id	12-15
Bufx			16-19
Offset			20-23
Sync			24-27
B_num			28-31
OS_Bufx			32-35
OS_Offset			36-39

Figure 8 – Schedule Header contents

6.1 Schedule Header fields

The Schedule Header fields shall be as follows. If an Operation does not use a particular Schedule Header field, then that field shall be transmitted as zeros.

Op (5 bits, high-order 5 bits of byte 00) – The Scheduled Transfer Operation. See tables 2 and 3 for a summary of Op values. Unspecified Op values are reserved.

Flags (11 bits, low-order 3 bits of byte 00, and all of byte 01) – Control flags (see 6.2).

S_count (16 bits, bytes 02-03):

– *In Request_Port, Request_Port_Response, and Request_State_Response Operations:* the number of available Slots (see 4.3.6);

– *In Request_To_Send_Response and Clear_To_Send Operations:* the Blocksize parameter (see 4.4.6);

– *In Data Operations:* the STU number (see 4.4.7).

R_Port (16 bits, bytes 04-05) – The receiver's logical Port for this Operation (see 4.3.2).

S_Port (16 bits, bytes 06-07) – The sender's logical Port for this Operation (see 4.3.2).

Key (32 bits, bytes 08-11) – Virtual Connection identifier. Generated independently by each end during the Virtual Connection setup. (See 4.3.3.)

R_id (16 bits, bytes 12-13)

– *In Request_To_Send_Response, Request_To_Receive_Response, Clear_To_Send, Request_State, Request_State_Response, End, and End_Ack Operations:* the receiver's Transfer identifier for this Operation (see 4.4.2).

– *In Request_To_Receive and Data Operations:* the Final Destination's Block identifier (B_id) for this Operation (see 4.4.3).

S_id (16 bits, bytes 14-15) – The sender's Transfer identifier for this Operation (see 4.4.2).

Bufx (32 bits, bytes 16-19):

– *In Request_Port and Request_Port_Response Operations:* the maximum buffer size (Bufsize) supported by the end device (see 4.3.4);

– *In Request_To_Receive, Clear_To_Send, and Data Operations:* the Buffer Index at the Final Destination (see 4.4.8).

Offset (32 bits, bytes 20-23):

– *In Request_Port and Request_Port_Response Operations:* the sender's Key value (see 4.3.3);

– *In Request_To_Receive, Clear_To_Send, and Data Operations:* the Final Destination's Offset within a Bufx (see 4.4.8);

– *In Request_State_Response Operations:* the Block number of the highest numbered contiguous Block received correctly (see 4.4.5).

Sync (32 bits, bytes 24-27):

– *In Request_Port and Request_Port_Response Operations:* the Max-STU size (see 4.3.5);

– *In Request_To_Send and Request_To_Receive Operations:* the high-order portion of the length, in bytes, of the Transfer data (see 4.4.4);

– *In Clear_To_Send Operations:* the high-order 16 bits shall be transmitted as zeros and the low-order 16 bits shall contain the Block identifier (B_id) (see 4.4.3).

– *In Data, Request_State, and Request_State_Response Operations:* the Sync parameter (see 4.3.6).

B_num (32 bits, bytes 28-31):

– *In Request_Port Operations:* the EtherType parameter (see 4.3.2);

– *In Request_To_Send, Request_To_Send _Response, and Request_To_Receive Operations:* the low-order portion of the length, in bytes, of the Transfer data (see 4.4.4)*;*

– *In Clear_To_Send and Data Operations:* the Block number being requested or transmitted (see 4.4.5);

– *In Request_State and Request_State_Response Operations:* the Block number being queried or responded to (see 4.4.5, 8.7, and 8.8).

OS_Bufx (32 bits, bytes 32-35):

– *In Request_To_Receive Operations:* the Buffer Index at the Originating Source (see 4.4.9);

– *In Data Operations:* Opaque data (see 4.4.10).

OS_Offset (32 bits, bytes 36-39):

– *In Request_To_Receive Operations:* the Originating Source's Offset within a Bufx (see 4.4.9);

– *In Clear_To_Send Operations:* the Final Destination's initial Offset value (see 4.4.8);

– *In Data Operations:* Opaque data (see 4.4.10).

6.2 Scheduled Transfer flags

Figure 9 summarizes the flags, and shows their relative position. The flag functions are detailed below for the case where the bit = 1.

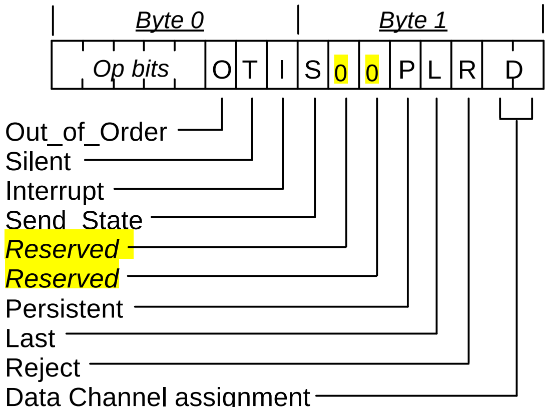


Figure 9 – Flags summary

Out_of_Order (b'1xxxxxxxxxx') = The end device is able to send and receive Blocks in any order.

Silent (b'x1xxxxxxxxx') = Requests silent delivery of a Data Operation. For Control Operations the Silent flag shall be ignored (i.e., transmitted as zero, but not checked at the receiver). For Data Operations with Silent = 1 the data transfer to the Destination Bufx is carried out normally, but the Schedule Header shall not be delivered to any upper-layer entity. This provides the basis for remote memory write semantics where the intent is to modify the contents of a remote memory without executing software in the Destination host computer.

Interrupt (b'xx1xxxxxxxx') = Requests that a signal or interrupt be generated and delivered to the appropriate upper-layer entity. The Interrupt flag is independent of the Silent flag, i.e., Interrupt = 1 calls for a signal whether or not Silent = 1. (See 4.5.5.)

NOTE 1 – The Silent and Interrupt flags together provide for three delivery modes for Data Operations: silent, polled, or interrupt-driven. If Silent = 1, the data payloads are delivered silently. If Silent = 0, then the upper-layer entity is informed by the same means used for all other Schedule Headers. This mode is suitable for polled interfaces. If Interrupt = 1, then a signal is delivered.

Send_State (b'xxx1xxxxxxx') = Requests that the Final Destination respond with a Request_State_Response upon successful receipt of this STU, or Operation, by the higher-layer protocol. Send_State is always valid on Control Operations. For Send_State to be valid on a Data Operation, either Interrupt = 1 or Silent = 0 must be true.

Reserved (b'xxxx00xxxxx') = The reserved flag bits shall be transmitted as zeros.

Persistent (b'xxxxxx1xxxx') = Retain the Final Destination's buffers (see 4.3.7).

Last (b'xxxxxxx1xxx') = The last STU of a Block.

Reject (b'xxxxxxxx1xx') = The request (i.e., Request_Port, Request_To_Send, or Request_To_Receive) has been rejected.

Data Channel assignment: The Data Channel to be used to carry Data Operations. The Data Channel value is assigned in a Request_To_Send Operation and is the Data Channel to be used for Data Operations associated with this Transfer.

b'xxxxxxxxx01' = Data Channel 1
b'xxxxxxxxx10' = Data Channel 2
b'xxxxxxxxx11' = Data Channel 3

The maximum STU size sent on Data Channels 1 and 2 shall be 2^{17} bytes (i.e., 128 Kbytes). The maximum STU size sent on Data Channel 3 shall be 2^{31} bytes (i.e., 2 gigabytes).

NOTE 2 – Data Channel assignment value b'00' is reserved.

7 Virtual Connection management

In this clause, a Virtual Connection is set up between two Ports (see 4.3.2), called the A-Port and B-Port. The device that initiates the Virtual Connection is called device A, and the device at the other end is called device B.

In addition to the Port values, each Port shall assign and associate a Key value (A-Key and B-Key) with the Virtual Connection (see 4.3.3). Other parameters exchanged during the Virtual Connection setup include Buffer sizes (A-Bufsize and B-Bufsize, see 4.3.4), maximum STU sizes (Max-STU, see 4.3.5), and the number of available Slots (A-Slots and B-Slots, see 4.3.6). The end devices also inform each other of their capability to support Persistent (see 4.3.7), and out-of-order Block delivery (see 4.4.5).

The Operations used to set up and tear down Virtual Connections are detailed below and summarized in table 2. Only the fields used in each Operation are listed; all of the other Schedule Header fields shall be transmitted as zeros. While a particular field usually carries the parameter of the same name, fields sometimes carry other parameter values. In the Operations below, the specific parameter used in the Operation is listed first, and if it is not carried in the field of the same name, then the field name is included in square brackets.

7.1 Request_Port

Request_Port shall be used to set up a Virtual Connection between end device A and end device B.

Semantics – Request_Port (
Op,
Flags,
A-Slots [S_count],
B-Port [R_Port],
A-Port [S_Port],
A-Bufsize [Bufx],
A-Key [Offset],
A-Max-STU [Sync],
EtherType [B_num])

Op = x'01'

Flags (see 6.2) shall specify the Out_of_Order and Persistent flags. A value of 1 shall indicate that A supports that feature. The appropriate value for the Interrupt flag shall also be carried (see 4.5.5).

A-Slots, carried in the S_count field, shall specify the maximum number of Slots allocated in A for this Virtual Connection (see 4.3.6).

B-Port, carried in the R_Port field, shall specify B's logical Port value for this Virtual Connection. B-Port may be either the well-known Port (B will assign the Port value), or a peer Port, that provides the service (see 4.3.2).

A-Port, carried in the S_Port field, shall specify A's logical Port value for this Virtual Connection (see 4.3.2).

A-Bufsize, carried in the Bufx field, shall specify A's buffer size (see 4.3.4).

A-Key, carried in the Offset field, shall specify A's Key value for this Virtual Connection (see 4.3.3).

A-Max-STU, carried in the Sync field, shall be ≤ A-Bufsize when sent by A (see 4.3.5). The A-Max-STU value received by B shall be used by B as the maximum size of STUs (Max-STU) to be sent from B to A on this Virtual Connection. (See 4.3.5.)

EtherType, carried in the B_num field, shall be a value that characterizes the ULP data payloads that will be exchanged on this Virtual Connection (see 4.3.2).

Issued – By device A.

Effect – If it accepts the request, then end device B shall establish a Virtual Connection and shall reply with a Request_Port_Response Operation. If rejected, then end device B shall respond with Reject = 1 in the Request_Port_Response (see 4.5.3).

7.2 Request_Port_Response

Request_Port_Response shall inform end device A whether the Virtual Connection was accepted or not. If accepted, the parameters associated with this Virtual Connection are passed to A.

Semantics – Request_Port_Response (
Op,
Flags,
B-Slots [S_count],
A-Port [R_Port],
B-Port [S_Port],
A-Key [Key],
B-Bufsize [Bufx],
B-Key [Offset],
B-Max-STU [Sync])

Op = x'02'

Flags (see 6.2) shall specify the Out_of_Order and Persistent flags. A value of 1 shall indicate that B supports that feature. The appropriate value for the Reject and Interrupt flags shall also be carried (see 4.5.3 and 4.5.5).

B-Slots, carried in the S_count field, shall specify the maximum number of Slots allocated in B for this Virtual Connection (see 4.3.6).

A-Port, carried in the R_Port field, shall be the same as the A-Port value in the Request_Port Operation (see 4.3.2).

B-Port, carried in the S_Port field, shall specify B's logical Port value for this Virtual Connection (see 4.3.2).

A-Key, carried in the Key field, shall be the Key value assigned by A in the Request_Port Operation (see 4.3.3).

B-Bufsize, carried in the Bufx field, shall specify B's buffer size (see 4.3.4).

B-Key, carried in the Offset field, shall specify B's Key value assigned for this Virtual Connection (see 4.3.3).

B-Max-STU, carried in the Sync field, shall be ≤ B-Bufsize when sent by B (see 4.3.5). The B-Max-STU value received by A shall be used by A as the maximum size of STUs (Max-STU) to be sent from A to B on this Virtual Connection. (See 4.3.5.)

Issued – By B in response to a Request_Port.

Effect – End device A has been assigned a logical Port on end device B. The Ports, Keys, buffer sizes, maximum STU size, and maximum number of Slots have been exchanged, and a Virtual Connection has been established. Note that the Virtual Connection is bi-directional in that either A or B may initiate a Scheduled Transfer. Multiple Scheduled Transfers may occur over a single Virtual Connection, and the Scheduled Transfers can be either writes or reads.

7.3 Port_Teardown

Port_Teardown shall terminate the Virtual Connection and may be issued by either end device A or end device B. The Port_Teardown sequence uses a three-way handshake consisting of Port_Teardown, Port_Teardown_Ack, and Port_Teardown_Complete that decreases timeout dependency for releasing resources.

Open Issue – A state table describing the 3-way handshake will be included in a Normative annex.

Semantics – Port_Teardown (
Op,
Flags,
R_Port,
S_Port,
Key)

Op = x'03'

Flags shall contain the appropriate value for the Interrupt flag (see 4.5.5).

R_Port shall contain the value associated with the receiver of the Operation, e.g., R_Port = B-Port when the Port_Teardown is issued by A (see 4.3.2).

S_Port shall contain the value associated with the sender of the Operation, e.g., S_Port = A-Port when the Port_Teardown is issued by A (see 4.3.2).

Key shall contain the Key value associated with the receiver of the Operation, e.g., Key = B-Key when the Port_Teardown is issued by A (see 4.3.3).

Issued – By either side, i.e., end device A or end device B, of the Virtual Connection. The sender should only issue a Port_Teardown when the Transfers are complete or appear to be stalled.

Effect – The receiver should release any buffers associated with this Virtual Connection, but shall retain the Port and Key values for use in further Port_Teardown Operations. The receiver shall also respond with a Port_Teardown_Ack.

7.4 Port_Teardown_Ack

Port_Teardown_Ack shall be used to acknowledge receipt of a Port_Teardown.

Semantics – Port_Teardown_Ack (
Op,
Flags,
R_Port,
S_Port,
Key)

Op = x'04'

Flags shall contain the appropriate value for the Interrupt flag (see 4.5.5).

R_Port shall contain the value associated with the receiver of the Operation, e.g., R_Port = B-Port when the Port_Teardown_Ack is issued by A (see 4.3.2).

S_Port shall contain the value associated with the sender of the Operation, e.g., S_Port = A-Port when the Port_Teardown_Ack is issued by A (see 4.3.2).

Key shall contain the Key value associated with the receiver of the Operation, e.g., Key = B-Key when the Port_Teardown_Ack is issued by A (see 4.3.3).

Issued – By the receiver of a Port_Teardown Operation after releasing this Virtual Connection's buffers.

Effect – The receiver should release any buffers associated with this Virtual Connection, but shall retain the Port and Key values for use in further Port_Teardown Operations. The receiver shall also respond with a Port_Teardown_Complete.

7.5 Port_Teardown_Complete

Port_Teardown_Complete shall be used to complete a three-way handshake, acknowledging that the actions associated with a Port_Teardown have been completed.

Semantics – Port_Teardown_Complete (
Op,
Flags,
R_Port,
S_Port,
Key)

Op = x'05'

Flags shall contain the appropriate value for the Interrupt flag (see 4.5.5).

R_Port shall contain the value associated with the receiver of the Operation, e.g., R_Port = B-Port when the Port_Teardown_Complete is issued by A (see 4.3.2).

S_Port shall contain the value associated with the sender of the Operation, e.g., S_Port = A-Port when the Port_Teardown_Complete is issued by A (see 4.3.2).

Key shall contain the Key value associated with the receiver of the Operation, e.g., Key = B-Key when the Port_Teardown_Complete is issued by A (see 4.3.3).

Issued – By the receiver of a Port_Teardown_Ack Operation.

Effect – After the Op_timeout expires twice, both the sender and receiver shall release the Virtual Connection's Port and Key values. The delay allows for lost or damaged Port_Teardown Operations to be re-issued.

8 Data movement

The Operations used for Scheduled Transfers are detailed below and summarized in table 3. All of the Scheduled Transfer data transfer Operations use the R_Port, S_Port, A-Key, and B-Key values that were assigned during the Virtual Connection setup (see 7.1 and 7.2). When end device A issues the Operation:

R_Port = B-Port
S_Port = A-Port
Key = B-Key

Likewise, when end device B issues the Operation:

R_Port = A-Port
S_Port = B-Port
Key = A-Key

For clarity and brevity, these values are not discussed in the individual Operations. All other Schedule Header fields that are not listed in a specific Operation shall be transmitted as zeros. While a particular field usually carries the parameter of the same name, fields sometimes carry other parameter values. In the Operations below, the specific parameter used in the Operation is listed first, and if it is not carried in the field of the same name, then the field name is included in square brackets.

8.1 Request_To_Send

Request_To_Send asks that space be allocated, and authorization be given, for a Transfer. Request_To_Send is issued by the Originating Source to specify the number of data bytes to be sent from the Originating Source to the Final Destination. In addition, the Originating Source shall specify whether 64-bit address or Buffer Indexes are used, whether the Final Destination's buffer should be persistent or discarded after a Block, and the Data Channel assignment for the data transfer. Note that the end device on either end of the Virtual Connection may issue a Request_To_Send. A Request_To_Send, with Persistent = 1, is also used to set up and expose memory for Request_To_Receive Operations.

Semantics – Request_To_Send (
Op,
Flags,
R_Port,
S_Port,
Key,
S_id,
T_len [Sync,B_num])

Op = x'16'

Flags (see 6.2) shall specify the Persistent, and Data Channel assignment flags. Persistent shall only = 1 if the corresponding flag was set = 1 by the other end during the Virtual Connection setup (see 7.1 and 7.2) and the function is desired for this Operation. The appropriate value for the Interrupt flag shall also be carried (see 4.5.5).

S_id shall be the Originating Source's Transfer identifier (see 4.4.2) used to identify this Transfer. The Final Destination shall use this value as the R_id parameter when replying to the Originating Source concerning this Transfer.

T_len, carried in the concatenation of the Sync and B_num fields, shall specify the total number of data payload bytes in the Transfer or that the size of the Transfer is unlimited (see 4.4.4).

Issued – By the Originating Source after a Virtual Connection has been established.

Effect – If rejected, the Final Destination will respond with a Request_To_Send_Response (see 8.2) with Reject = 1. If accepted, the Final Destination will respond with a Request_To_Send_Response with Reject = 0. If accepted, the Final Destination will set up its memory region for the Transfer and respond with one or more Clear_To_Send Operations (see 8.5) when the memory region is ready. The Final Destination may omit sending the Request_To_Send_Response if the Clear_To_Send Operation can be returned before the timeout (see 9.1) on the Request_To_Send Operation expires.

8.2 Request_To_Send_Response

Request_To_Send_Response shall inform the Originating Source whether the Transfer was accepted or not. If accepted, the Request_To_ Send_Response specifies the Transfer identifier (see 4.4.2) assigned by this end (i.e., the Final Destination) for this Transfer and the number of STUs per Block (see 4.4.7). A Request_To_Send_Response does not give the Originating Source permission to start sending; that comes from a Clear_To_Send. A Clear_To_Send may be used instead of a Request_To_Send_Response if the Final Destination is able to immediately accept the data.

Semantics – Request_To_Send_Response (
Op,
Flags,
Blocksize [S_count],
R_Port,
S_Port,
Key,
R_id,
S_id)

Op = x'17'

Flags (see 6.2) shall specify the Reject and Interrupt flags (see 4.5.3 and 4.5.5).

Blocksize, carried in the S_count field, shall specify the Block size (see 4.4.6).

R_id shall be the Transfer identifier (see 4.4.2) assigned by the Originating Source in the Request_To_Send Operation.

S_id shall be the Transfer identifier (see 4.4.2) used by the Final Destination to identify this Transfer. The Originating Source shall use this value as the R_id parameter when replying to the Final Destination concerning this Transfer.

Issued – By the Final Destination.

Effect – The Originating Source shall segment the Transfer into Blocks and STUs for transmission.

8.3 Request_To_Receive

Request_To_Receive, issued by the Final Destination (the initiator), asks for a single Block of data to be sent from a previously allocated location in the Originating Source. The Request_To_Receive specifies the number of data bytes to be sent from the Originating Source to the Final Destination. A Request_To_Receive transfers a single Block; there is no notion of a multi-Block Request_To_Receive data movement.

When a Request_To_Receive is issued, it is assumed that the ULPs on both end devices had previously allocated resources for the entire Transfer through a previous Request_To_Send Operation. Note that the device at either end of the Virtual Connection may issue a Request_To_Receive.

Request_To_Receive may be used in conjunction with Persistent memory. The Request_To_Receive initiator must first request a remote Persistent memory region by issuing a Request_To_Send Operation with Persistent = 1. The following associated Clear_To_Send Operation, and possibly Request_To_Send_Response Operation (see 8.1), establish an R_id / S_id pair which together identify the Transfer. The persistent memory for the Transfer remains available until the Transfer is terminated by either an End / End_Ack exchange or Port_Teardown sequence.

If accepted, an (optional) Request_To_Send-_Response will be returned with Persistent = 1. This will be followed by a Clear_To_Send Operation which establishes the memory region dedicated for the Transfer. The Request_To_Send / Clear_To_Send handshake establishes an R_ID, S_ID pair which together identify the "Transfer". The Persistent memory for the Transfer remains available until the Transfer is terminated by either an End/End_Ack exchange or Port_Teardown sequence.

While a Persistent Transfer is active, it is available for an unlimited number of Data

Operations or Request_To_Receive Operations as long as the number of outstanding Operations at any time falls within the limits established by the Slot mechanism (section 4.3.6).

Semantics – Request_To_Receive (
Op,
Flags,
R_Port,
S_Port,
Key,
R_id,
S_id,
Bufx,
Offset,
T_len [Sync,B_num],
OS_Bufx,
OS_Offset)

Op = x'18'

Flags (see 6.2) shall specify the Interrupt flag (see 4.5.5).

B_id shall be the Block identifier (see 4.4.3) being assigned by this Operation. B_id shall be placed in the low-order 16 bits of the Sync field; the high-order bits shall be transmitted as zeros.

S_id shall be the Transfer identifier (see 4.4.2) used by the Final Destination to identify this Transfer. The Originating Source shall use this value as the R_id parameter when replying to the Final Destination concerning this Transfer.

Bufx shall specify the initial Buffer Index in the Final Destination where the data will be placed (see 4.4.8).

Offset is a value that the Final Destination must receive with the first STU of the Block so that the data can be properly placed in the Final Destination's memory (see 4.4.8).

T_len, carried in the concatenation of the Sync and B_num fields, shall specify the total number of data payload bytes in the Transfer (see 4.4.4).

OS_Bufx shall specify the Originating Source's Buffer Index (see 4.4.9).

OS_Offset shall specify the Originating Source's offset value (see 4.4.9).

Issued – By the Final Destination.

Effect – If accepted, the Originating Source shall send the specified Block of data. If rejected, the Originating Source shall reply with Reject = 1 in a Request_To_Receive_Response (see 4.5.3).

8.4 Request_To_Receive_Response

Request_To_Receive_Response, may be issued by the Originating Source in response to a Request_To_Receive Operation. The Originating Source may omit sending the Request_To_Receive_Response if the associated Data Operation can be returned before the timeout (see 9.1) on the Request_To_Receive Operation expires.

Semantics – Request_To_Receive_Response (
Op,
Flags,
R_Port,
S_Port,
Key,
R_id,
S_id)

Op = x'19'

Flags (see 6.2) shall specify the Reject and Interrupt flags.

R_id shall be the Transfer identifier (see 4.4.2) assigned by the Originating Source in the Request_To_Send Operation.

S_id shall be the Transfer identifier (see 4.4.2) used by the Final Destination to identify this Transfer.

Issued – By the Originating Source.

Effect – The Request_To_Receive Operation has been rejected (see 4.5.3), or the associated Data Operation will be delayed.

8.5 Clear_To_Send

Clear_To_Send shall be used to give the Originating Source permission to send one Block. Clear_To_Send may also be used to request retransmission of a Block from systems that are capable of retransmission.

Semantics – Clear_To_Send (
Op,
Flags,
Blocksize [S_count],
R_Port,
S_Port,
Key,
R_id
S_id,
Bufx,
Offset,
B_id [*,Sync],
B_num,
I_Offset [OS_Offset])

Op = x'1A'

Flags (see 6.2) shall specify the Interrupt flag (see 4.5.5).

Blocksize, carried in the S_count field, shall specify the Block size (see 4.4.6).

R_id shall be the Transfer identifier (see 4.4.2) assigned by the remote end (the Originating Source) of the Virtual Connection.

S_id shall be the Transfer identifier (see 4.4.2) assigned by this end (the Final Destination) of the Virtual Connection.

Bufx shall specify the initial Buffer Index in the Final Destination where the data will be placed (see 4.4.8).

Offset is a value that the Final Destination must receive with the first STU of a Block so that the data can be properly placed in the Final Destination's memory (see 4.4.8).

B_id shall be the Block identifier (see 4.4.3) being assigned by this Operation for this Block of the Transfer. B_id shall be placed in the low-order 16 bits of the Sync field; the high-order bits shall be transmitted as zeros.

B_num shall be the Block number being given permission to be transmitted (see 4.4.5).

I_Offset, carried in the OS_Offset field, shall specify the Offset value associated with the first Block of the Transfer (see 4.4.8).

Issued – By the Final Destination.

Effect – The Originating Source shall send the specified Block.

8.6 Data

A Data Operation sends an STU of a Block from the Originating Source to the Final Destination. No STU shall be larger than the maximum STU size determined during the Virtual Connection setup (see 7.2).

Semantics – Data (
Op,
Flags,
S_count,
R_Port,
S_Port,
Key,
B_id,
S_id,
Bufx,
Offset,
Sync,
B_num,
Opaque [OS_Bufx],
Opaque [OS_Offset])

Op = x'1B'

Flags (see 6.2) shall specify the Interrupt, Silent, Send_State, Last, and Data Channel assignment flags (see 6.2). Send_State may be sent with any STU of a Block. The Request_State_Response Control Operation associated with this request shall be issued after processing this STU when Send_State = 1. The sender shall copy (in this Data Operation) the Data Channel assignment flags supplied in the corresponding Request_To_Send Operation; the value is a do not care at the receiver.

S_count shall be the STU number (see 4.4.7).

B_id shall be the Block identifier (see 4.4.3) assigned by the other end (the Final Destination) of the Virtual Connection.

S_id shall be the Transfer identifier (see 4.4.2) assigned by this end of the Virtual Connection. Note that if this is the first STU associated with a Request_To_Receive Operation, then this Transfer identifier (see 4.4.2) is being assigned by the Originating Source and shall be used by the Final Destination as the R_id parameter when replying to the Originating Source concerning this Transfer.

Bufx shall be the Buffer Index at the Final Destination (see 4.4.8).

Offset shall be the Final Destination's offset within a Bufx (see 4.4.8).

Sync shall be a value assigned by the Originating Source to synchronize the current view of the number of empty Slots in the Final Destination (see 4.3.6).

B_num shall be the number of the Block that this STU is a part of.

Opaque data, carried in the OS_Bufx and OS_Offset fields, shall be as specified in 4.4.10.

Issued – By the Originating Source.

Effect – The Final Destination shall place the STU data in the memory area pointed to by Bufx and offset by the Offset value. The Final Destination shall only accept data into pre-allocated buffer regions. The Final Destination is responsible for ensuring that all of the Blocks of a Transfer are received. The actions to be taken if a Block is missing are beyond the scope of this standard.

8.7 Request_State

Request_State is used to request that the remote end device provide its current number of empty Slots for Schedule Headers, the Block number associated with the last set of contiguously good data received, and whether the named Block was received correctly.

Semantics – Request_State (
Op,
Flags,
R_Port,
S_Port,
Key,
R_id,
S_id,
Sync,
B_num)

Op = x'1C'

Flags shall contain the appropriate value for the Interrupt flag (see 4.5.5).

R_id shall be the Transfer identifier (see 4.4.2) assigned by the remote end device of this Virtual Connection. R_id = x'0000' means that the receiver shall not look for a current Transfer and only return the current number of empty Slots for this Virtual Connection.

S_id shall be the Transfer identifier (see 4.4.2) assigned by this end of the Virtual Connection. If R_id = x'0000', then S_id shall also be x'0000'.

Sync shall be a value assigned by the local end device (sender) to synchronize the current view of the number of empty Slots in the remote end device (receiver). (See 4.3.6.)

B_num shall indicate the Block number being queried. B_num = x'FFFFFFFF' indicates that the sender does not care about the status of any particular Block.

Issued – By an end device that needs state information from the remote end device of the Virtual Connection. The sender may not have received the Request_State_Response that it expected from a Data Operation and can send a Request_State to recover from a lost or damaged Request_State_Response.

Effect – The receiver shall reply with a Request_State_Response.

8.8 Request_State_Response

Request_State_Response shall be used to indicate the number of empty Slots in this Port of the Virtual Connection (see 4.3.6). Request_State_Response may also indicate the highest numbered contiguous Block received correctly and whether the Block indicated in the B_num parameter was received correctly (see 4.5.2).

Semantics – Request_State_Response (
Op,
Flags,
C-Slots [S_count],
R_Port,
S_Port,
Key,
R_id,
S_id,
B_seq [Offset],
Sync,
B_num)

Op = x'1D'

Flags (see 6.2) shall specify the Interrupt flag (see 4.5.5).

C-Slots, carried in the S_count field, shall indicate the sender's view of the number of empty Slots it has available for additional Operations on this Virtual Connection. (See 8.6.) C-Slots = x'FFFF' (i.e., -1) shall indicate that this end device does not implement the Slots mechanism for Operations flow control.

R_id shall echo the S_id value in the Request_State or Data Operation that triggered this Request_State_Response.

S_id shall echo the R_id value in the Request_State or Data Operation that triggered this Request_State_Response. S_id = x'0000' shall mean that the B_seq and B_num parameters are meaningless.

B_seq, carried in the Offset field, shall indicate the highest numbered contiguous Block received correctly. B_seq = x'FFFFFFFF' shall indicate that no Transfers are in progress or no Blocks have been received.

Sync is echoed from the Request_State, or Data Operation with Send_State = 1, that initiated this Request_State_Response Operation (see 4.3.6).

B_num shall echo the Block number, carried in the B_num field of the Data Operation or the Request_State Operation, if the indicated Block was received correctly. If the indicated Block has not been correctly received, then B_num shall contain x'FFFFFFFF'. The Sync value can be used by the receiving end to identify the Operation containing the B_num being queried.

Issued – It is intended that a Request_State_Response be issued by an end device's ULP after receiving Send_State = 1 in a Data Operation, or after receiving a Request_State Operation, or to reject an Operation.

Effect – State information is passed to the other end of the Virtual Connection.

8.9 End

End allows either end of the Virtual Connection to terminate a Scheduled Transfer before it has completed and to terminate a Scheduled Transfer of unlimited size.

Semantics – End (
Op,
Flags,
R_Port,
S_Port,
Key,
R_id
S_id)

Op = x'1E'

Flags shall contain the appropriate value for the Interrupt flag (see 4.5.5).

R_id shall be the Transfer identifier (see 4.4.2) assigned by the other end of the Virtual Connection.

S_id shall be the Transfer identifier (see 4.4.2) assigned by this end of the Virtual Connection. This S_id value shall not be reused until an End_Ack is received.

Issued – By the Originating Source or the Final Destination.

Effect – A Final Destination receiving an End shall stop sending Control Operations associated with this Scheduled Transfer. An Originating Source receiving an End shall stop sending Control Operations and STUs associated with this Scheduled Transfer. An End kills a Scheduled Transfer, but shall not affect the Virtual Connection carrying the Scheduled Transfer.

Table 2 – Virtual Connection Operations summary between end devices A and B

	Op	Flags	S_count	R_Port	S_Port	Key	Bufx	Offset	Sync	B_num
RQP	x'01'	***OIP***	***A-Slots***	***B-Port***	***A-Port***	*	***A-Bufsize***	***A-Key***	***A-Max-STU***	***EtherType***
RQPR	x'02'	***OIPR***	***B-Slots***	A-Port	***B-Port***	A-Key	***B-Bufsize***	***B-Key***	***B-Max-STU***	*
PT	x'03'	***I***	*	R_Port	S_Port	R-Key	*	*	*	*
PTA	x'04'	***I***	*	R_Port	S_Port	R-Key	*	*	*	*
PTC	x'05'	***I***	*	R_Port	S_Port	R-Key	*	*	*	*

NOTES –

1 – Operation abbreviations:

PT = Port_Teardown
PTA = Port_Teardown_Ack
PTC = Port_Teardown_Complete
RQP = Request _Port
RQPR = Request_Port_Response

2 – Flag abbreviations are: O = Out_of_Order, I = Interrupt, P = Persistent, R = Reject

3 – R-Key = Key value the receiver binds to, e.g., R-Key = A-Key when Operation issued by device B.

4 – R_Port = Port number in device receiving the Operation, e.g., R_Port = A-Port when issued by device B.

5 – S_Port = Port number in device sending the Operation, e.g., S_Port = B-Port when issued by device B.

6 – The Schedule Header fields that are not shown shall be transmitted as zeros.

SYMBOLS -

* = Unused value, transmit as 0

Values in bold italics are assigned by the specific Operation and may be used by later Operations

8.10 End_Ack

End_Ack confirms that the sending end device has seen and acted on the End.

Semantics – End_Ack (
Op,
Flags,
R_Port,
S_Port,
Key,
R_id,
S_id)

Op = x'1F'

Flags shall contain the appropriate value for the Interrupt flag (see 4.5.5).

R_id shall be the Transfer identifier (see 4.4.2) assigned by the remote end device of this Virtual Connection.

S_id shall be the Transfer identifier (see 4.4.2) assigned by this end of the Virtual Connection. This S_id value should not be immediately reused to avoid aliasing.

Issued – By the end of the Virtual Connection that received the End Operation.

Effect – Acknowledgment that the Scheduled Transfer has been terminated.

Table 3 – Data transfer and status Operations summary between end devices S and R

	Op	Flags	S_count	R_id	S_id	Bufx	Offset	Sync	B_num	OS_Bufx	OS_Offset
RTS	x'16'	***IPD***	*	*	***S_id***	*	*	***T_len***		*	*
RTSR	x'17'	***IR***	***Blocksize***	R_id	***S_id***	*	*	*	*	*	*
RTR	x'18'	***I***	*	***B_id***	S_id	***Bufx***	***Offset***	***T_len***		***OS_Bufx***	***OS_Offset***
RTRR	x'19'	***IR***	*	R_id	S_id	*	*	*	*	*	*
CTS	x'1A'	***I***	***Blocksize***	R_id	S_id	***Bufx***	***Offset***	*, ***B_id***	***B_num***	*	***I_Offset***
Data	x'1B'	***ITSL***D	***S_count***	B_id	S_id	***Bufx***	***Offset***	***Sync***	B_num	***Opaque***	
RS	x'1C'	***I***	*	R_id	S_id	*	*	***Sync***	***B_num***	*	*
RSR	x'1D'	***IR***	***C-Slots***	R_id	S_id	*	***B_seq***	Sync	B_num	*	*
End	x'1E'	***I***	*	R_id	S_id	*	*	*	*	*	*
EndA	x’1F’	***I***	*	R_id	S_id	*	*	*	*	*	*

NOTES –

1 – Operation abbreviations:

CTS = Clear_To_Send
EndA = End_Ack
RS = Request_State
RSR = Request_State_Response
RTR = Request_To_Receive
RTRR = Request_To_Receive_Response
RTS = Request_To_Send
RTSR = Request_To_Send_Response

2 – Flag abbreviations : I = Interrupt, T = Silent, S = Send_State, P = Persistent, L = Last STU of Block, R = Reject, D = Data Channel assignment

3 – R_id = Transfer identifier in device receiving the Operation, e.g., R_id = G_id when issued by device H.

4 – S_id = Transfer identifier in device sending the Operation, e.g., S_id = H_id when issued by device H.

5 – Schedule Header parameters that shall be transmitted as assigned in RQP and RQPR Operations:

R_Port = Port number of the device receiving the Operation
S_Port = Port number of the device sending the Operation
Key = Key value assigned by the device receiving the Operation

SYMBOLS -

* = Unused value, transmit as 0

Values in bold italics are assigned by the specific Operation and may be used by later Operations

9 Error processing

Table 5 is a summary of the logged errors. The logging is on a per-Port basis, and shall be available to the ULP that is using the Port. The nature and size of the logs are system dependent.

9.1 Operation timeout

Errors other than syntactic errors are manifested as missing Operations, occurring when the underlying physical media discard or damage a transmission (see 4.5.4). Such errors are detected by Op_timeout, which is system and/or Port dependent. Op_timeout_Occurances shall be logged. Example means for determining the Op_timeout value for a Virtual Connection include:

– a time longer than the measured round-trip time through the software path (use a Request_State / Request_State_Response pair to measure on a per-Port basis); or

– a long fixed time period.

Another system and/or Port dependent parameter, Max_Retry, specifies the maximum number of times to retry an Operation. If Max_Retry is reached without success, then the Operation is considered to be aborted and control shall be passed to the ULP. Max_Retry_Occurances shall be logged.

9.2 Operation Pairs

Table 4 lists the Operation pairs – command and response, or response and completion – that shall be retried if the associated response is not received within an Op_timeout.

In addition to the entries in table 4, Request_State_Response is a corresponding pair for Data Operations which have Send_State = 1. If the Request_State_Response is not received, then the Originating Source may send a Request_State to obtain the state information.

The ULP in the Final Destination that issues a Clear_To_Send, or Request_To_Receive, is responsible for timing out these Operations. The ULP may or may not use Op_timeout to indicate failure.

Table 4 – Operation pairs guarded by Op_timeout

Operation	Response(s)
Request_Port	Request_Port_Response
Port_Teardown	Port_Teardown_Ack
Port_Teardown_Ack	Port_Teardown_Complete
Request_To_Send	Request_To_Send_Response or Clear_To_Send
Request_To_Receive	Data or Request_To_Receive_Response
Request_State	Request_State_Response
End	End_Ack

9.3 Syntax errors

9.3.1 Undefined Opcode

An Operation with an undefined Opcode value shall be discarded, an Undefined_Opcode_Error shall be logged, and the Opcode logged in Undefined_Opcode_Value.

9.3.2 Unexpected Opcode

Most of the Operations require previous Operations to set up state on each device. If a device receives an out of sequence Opcode (e.g., receiving a Request_Port_Response without sending the initiating Request_Port), the Operation shall be discarded, an Unexpected_Opcode_Error shall be logged, and the Opcode logged in Unexpected_Opcode_Value.

9.4 Virtual Connection errors

9.4.1 Invalid Key or Port

All Operations, excluding Request_Port, should have a Key (see 4.3.3) value that validates the Operation for the Virtual Connection. Operations with an invalid Key shall not be executed, and an Invalid_Key_Error shall be logged.

All Operations should have a valid Destination Port value (see 4.3.2). Operations with an invalid Destination Port value shall not be executed, and an Invalid_Port_Error shall be logged.

NOTE – Multiple contiguous invalid Key and/or Port values may indicate a problem with the link or a malicious host on the network. The supervising process should be informed.

9.4.2 Slots exceeded

Operations that exceed the number of Slots (see 4.3.6) for the Virtual Connection may not be executed, and a Slots_Exceeded_Error shall be logged.

9.4.3 Unknown EtherType

If a Request_Port Operation contains an unknown EtherType (see 4.3.2), the receiver shall issue a Request_Port_Response with the Reject bit set and log an Unknown_EtherType_Error.

9.4.4 Illegal Bufsize

If a Request_Port contains a Bufsize (see 4.3.4) value that is < 8 or > 63, (i.e., Buffer size < 2^8 bytes, or > 2^{63} bytes), then the receiver shall respond with a Request_Port_Response with Reject = 1. If a Request_Port_Response contains a Bufsize value that is < 8 or > 63, then the receiver shall respond with a Port_Teardown. In either case, an Illegal_Bufsize_Error shall be logged.

9.4.5 Illegal STU size

The maximum STU sizes (A-Max-STU and B-Max-STU) were determined during the Virtual Connection setup (see 4.3.5, 7.1 and 7.2). If the received STU is greater than the maximum STU size, then the STU shall be discarded and an Illegal_STU_Size_Error shall be logged.

9.5 Scheduled Transfer errors

9.5.1 Invalid S_id

All Scheduled Transfer Operations, except Request_To_Send, should have a valid Destination id (R_id) (see 4.4.2) for quickly accessing state information for this Scheduled Transfer. After checking the R_id, the S_id should match the stored value for this Transfer. An invalid S_id shall result in not executing the Operation and logging an Invalid_S_id_Error.

9.5.2 Bad Data Channel specification

During a Request_To_Send Operation, the sending device declares the lower layer Data Channel that will carry Data Operations for this Scheduled Transfer. Some Data Channels may not be available for Scheduled Transfers depending on the lower layer (e.g., b'00' is not a valid choice on HIPPI-6400 as it indicates VC0 which is reserved for Control Operations). The receiver shall issue a Request_To_Send_Response with the Reject bit set.

9.5.3 Persistent not available

If the Virtual Connection did not specify the capability for Persistent (see 4.3.7) during the Virtual Connection establishment (see 7.1 and 7.2), any Scheduled Transfer Operations on this Virtual Connection with the Persistent bit set shall not be executed. The Operation shall be rejected and a Persistent_Error logged.

9.5.4 Out of Range B_num, Bufx, Offset, or S_count

During the Clear_To_Send, Data, and Request_State_Response Operations, a Block number (see 4.4.5) may appear that is outside the calculated number of Blocks for the Transfer. If an out of range Block number is encountered, the receiver shall not execute the Operation and shall log an Out_Of_Range_B_num_Error.

If a Data Operation contains a Bufx and/or Offset (see 4.4.8) that exceeds the buffer range allocated by the Final Destination for outstanding Clear_To_Sends, then the receiver shall not execute the Operation and shall log an Out_Of_Range_Bufx_Error.

If a Data Operation contains an Offset (see 4.4.8) larger than the buffer size, the receiver shall not execute the Operation and shall log an Oversized_Offset_Error.

If a Data Operation contains an S_count (see 4.4.7) that is not one greater than the previous STU for this Block, then the STU is out of order. The receiver shall discard the STU and log an Out_Of_Order_STU_Error.

9.5.5 Block out of order error

If a Data Operation contains a B_num that is not one greater than the previous B_num for this Transfer, and Out_of_Order (see 6.2) capability was not specified during the Virtual Connection establishment (see 7.1), then the Final Destination shall log an Out_Of_Order_B_num and may terminate the Transfer with an End sequence.

9.5.6 Illegal Blocksize

If a Request_To_Send_Response, or Clear_To_Send, contains a Blocksize (see 4.4.6) value that is < 8 or > 63, (i.e., Block size < 2^8 bytes, or > 2^{63} bytes), then the receiver should discard the offending Operation and log an Illegal_Blocksize_Error.

9.5.7 Request_To_Receive error

The Request_To_Receive Operation (see 8.3) must be set up by previous Request_To_Send and Clear_To_Send Operations. If these Operations have not occurred, then the Request_To_Receive Operation shall be discarded, a Request_To_Receive_Response with Reject = 1 sent to the remote end device in response, and a Request_To_Receive_Error logged.

9.5.8 Undefined Flag

If a received Operation contains a flag =1 and use of that flag is not defined for that Operation, then the flag shall be ignored and an Improper_Flag_Use_Error should be logged.

Table 5 – Summary of logged errors

Name	Occurs in Operation
Illegal_Blocksize_Error	RTSR, CTS
Illegal_Bufsize_Error	RQP, RQPR
Illegal_STU_Size_Error	Data
Improper_Flag_Use_Error	all
Invalid_Key_Error	all except RQP
Invalid_Port_Error	all
Invalid_S_id_Error	all with an R_id
Max_Retry_Occurance	End, PT, PTA, RQP, RS, RTR, RTS
Op_timeout_Occurance	End, PT, PTA, RQP, RS, RTR, RTS
Out_Of_Order_B_num	Data
Out_Of_Order_STU_Error	Data
Out_Of_Range_B_num_Error	CTS, Data, RS, RSR
Out_Of_Range_Bufx_Error	Data
Oversized_Offset_Error	Data
Persistent_Error	RTS
Request_To_Receive_Error	RTR
Slots_Exceeded_Error	all with Opcode ≥ 6
Undefined_Opcode_Error	not applicable
Undefined_Opcode_Value	not applicable
Unexpected_Opcode_Error	all except RQP
Unexpected_Opcode_Value	all except RQP
Unknown_EtherType_Error	RQP

Operation abbreviations:
CTS = Clear_To_Send
PT = Port_Teardown
PTA = Port_Teardown_Ack
RQP = Request_Port
RQPR = Request_Port_Response
RS = Request_State
RSR = Request_State_Response
RTR = Request_To_Receive
RTS = Request_To_Send
RTSR = Request_To_Send_Response

Annex A
(normative)

Using lower layer protocols

A.1 HIPPI-6400-PH as the lower layer

ANSI X3.xxx defines HIPPI-6400-PH, portions of which are repeated here as an aid to the reader. As shown in figure A.1, HIPPI-ST Operations shall be carried over HIPPI-6400-PH with the first eight bytes of the HIPPI-ST Schedule Header occupying the last eight bytes of the HIPPI-6400-PH Header micropacket.

All HIPPI-ST Control Operations shall be carried on HIPPI-6400-PH Virtual Channel VC0. Data Operations shall use Virtual Channel 1, 2, or 3 as specified in the HIPPI-ST Data Channel Assignment flag bits (see 6.2) and carried in a Request_To_Send Operation (see 8.1).

HIPPI-ST shall also specify the EtherType value that is placed in the HIPPI-6400-PH MAC Header (see the reference for RFC 1700 in 4.3.2).

M_len (in the HIPPI-6400-PH MAC Header), specifies the number of bytes following M_len, exclusive of any padding in the last micropacket. Hence, M_len will have the following values:

– M_len = 48 for Control Operations without an optional payload (i.e., 48 = 8 byte IEEE 802.2 LLC/SNAP Header + 40-byte HIPPI-ST Schedule Header);

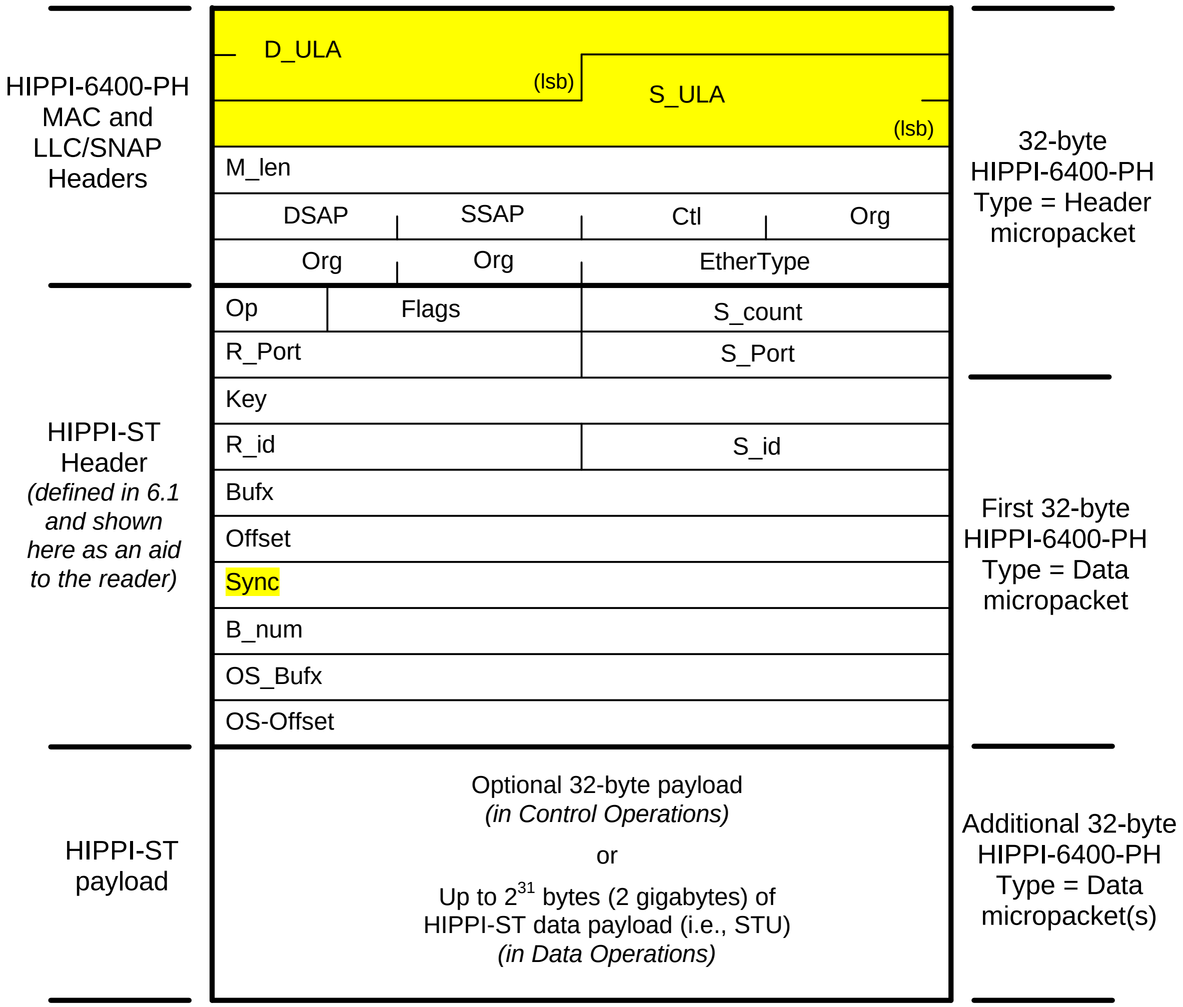
– M_len = 80 for Control Operations with optional payload;

– M_len = (48 + number of user data payload bytes) for Data Operations.

A.2 HIPPI-FP as the lower layer

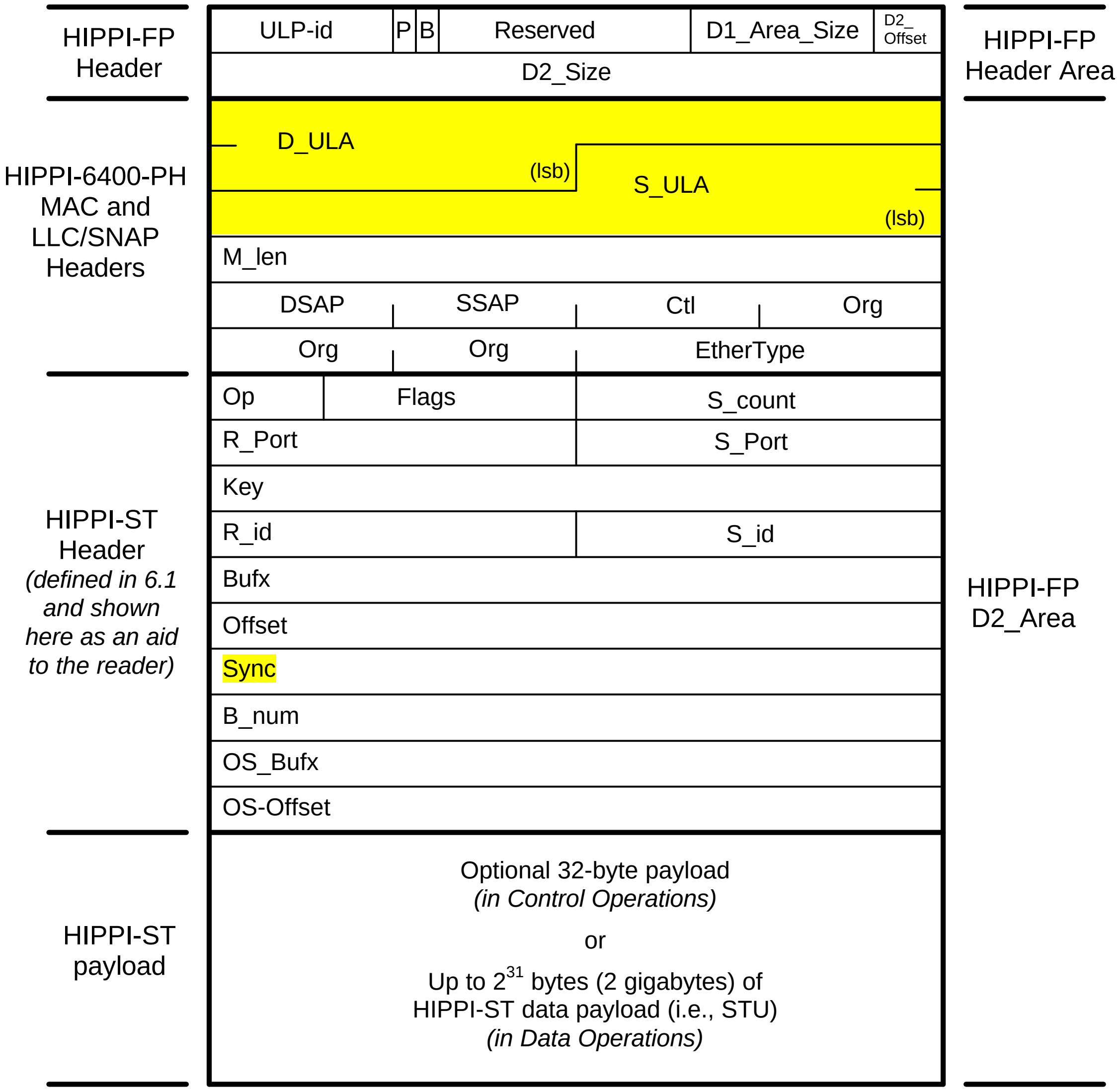
ANSI X3.210 defines HIPPI-FP, portions of which are repeated here as an aid to the reader. As shown in figure A.2, HIPPI-ST Operations shall be carried over HIPPI-FP in the D2_Area. The HIPPI-FP D1_Area shall not be used. The HIPPI-FP D2_Offset shall be set to zero. Short bursts shall only be used at the end of a packet, i.e., short first burst is disallowed. Note that D2_Size = M_len + 16.

The HIPPI-6400-PH MAC and LLC/SNAP Headers are defined in ANSI X3.xxx, portions of which are repeated here as an aid to the reader. The MAC and LLC/SNAP headers are included to facilitate translation to other protocols. The 48-bit ULA addresses allow address assignment and usage common to other networking technologies.


HIPPI-6400-PH MAC and LLC/SNAP Headers
D_ULA
(lsb)
S_ULA
(lsb)
M_len
DSAP
SSAP
Ctl
Org
Org
Org
EtherType
32-byte HIPPI-6400-PH Type = Header micropacket
HIPPI-ST Header *(defined in 6.1 and shown here as an aid to the reader)*
Op
Flags
S_count
R_Port
S_Port
Key
R_id
S_id
Bufx
Offset
Sync
B_num
OS_Bufx
OS-Offset
First 32-byte HIPPI-6400-PH Type = Data micropacket
HIPPI-ST payload
Optional 32-byte payload
(in Control Operations)
or
Up to 2^{31} bytes (2 gigabytes) of HIPPI-ST data payload (i.e., STU)
(in Data Operations)
Additional 32-byte HIPPI-6400-PH Type = Data micropacket(s)

NOTE – Shown as 32-bit words

Figure A.1 – HIPPI-ST Operations carried in HIPPI-6400-PH Messages


HIPPI-FP Header
ULP-id
P
B
Reserved
D1_Area_Size
D2_ Offset
HIPPI-FP Header Area
D2_Size
HIPPI-6400-PH MAC and LLC/SNAP Headers
D_ULA
(lsb)
S_ULA
(lsb)
M_len
DSAP
SSAP
Ctl
Org
Org
Org
EtherType
HIPPI-ST Header *(defined in 6.1 and shown here as an aid to the reader)*
Op
Flags
S_count
R_Port
S_Port
Key
R_id
S_id
Bufx
Offset
Sync
B_num
OS_Bufx
OS-Offset
HIPPI-FP D2_Area
HIPPI-ST payload
Optional 32-byte payload *(in Control Operations)*
or
Up to 2^{31} bytes (2 gigabytes) of HIPPI-ST data payload (i.e., STU) *(in Data Operations)*

NOTE – Shown as 32-bit words

Figure A.2 – HIPPI-ST Operations carried in HIPPI-FP packets

Annex B
(informative)

HIPPI-ST striping

B.1 Striping principles

HIPPI-ST is capable of supporting multiple physical interfaces for a single Transfer (see figures B.1–B.3). This striping capability may be of benefit when a single interface is not able to support required data rates. It may be especially useful where data is moved from many slower interfaces to a single faster interface or vice-versa. It may also be used with multiple interfaces at both the Originating Source and Final Destination. Mechanisms to set up, select, and control the underlying physical interfaces are beyond the scope of this standard.

The Block is the basic striping unit. Each Block contains sufficient information to completely identify an individual Transfer and the Block's location within the Transfer. The only difference between striped and non-striped operation is the selection of port MAC addresses to allow concurrent data movement. Striping is not done on an STU basis because striped STUs can not be guaranteed to be delivered in-order.

There are a few conventions that should be followed to facilitate striping:

– Block sizes (when striping is desirable) must be small enough to support concurrency and allow each channel to have at least one Block to send.

– Sufficient Clear_To_Send Operations should be kept outstanding by a data receiver to allow concurrent Data Operations.

– The interface adapter(s) must be capable of handling multiple Blocks simultaneously. This may require communication between interfaces (or their software drivers) within a system.

– The return physical address (e.g., ULA), for each Operation is specified by the Source ULA for that Operation. HIPPI-ST implementers should not assume that the Source ULA for a given port will remain constant.

– The Destination must signify that it supports delivery of Blocks in any order (i.e., Out_of_Order = 1, see 6.2) during the Virtual Connection setup.

B.2 Many-to-one striping

Figure B.1 shows using a number of lower-throughput interfaces, aggregated together, to communicate with one higher-throughput interface (using a translator or bridge). Striping the lower-throughput interfaces together can allow legacy systems to communicate quickly over newer network infrastructures. In this case, action to implement striping is required only on the side of the lower-throughput interface.

After port setup, data movement is initiated with a Request_To_Send Operation. A Request_To_Send_Response will be received, either as a discrete message or as part of a Clear_To_Send. As Clear_To_Send Operations are received, the system with multiple lower-throughput ports can move a Block of data for each Clear_To_Send received. As many Blocks can be in transit concurrently as there are ports to carry them and Clear_To_Send Operations authorizing them.

The system receiving these Blocks processes them normally, placing them into memory as their Bufx and Offset values dictate.

B.3 One-to-many striping

Figure B.2 shows how Transfers made from one higher-throughput interface can also be spread across more than one lower-throughput interface without any special action on the part of the higher-throughput system.

After port setup, the Transfer is initiated with a Request_To_Send Operation from the higher-throughput interface. The lower-throughput interface that has done the port setup will return a

Request_To_Send_Response, either as a discrete Operation or as part of a Clear_To_Send. Each Clear_To_Send issued should be sent from the interface desiring the data.

An alternative is to send all of the Clear_To_Send Operations from a single interface and substitute the desired physical return address (e.g., ULA) for the Clear_To_Send's Source ULA (making it appear that the Clear_To_Send's Source ULA was generated by the interface desiring the data). Subsequent Data Operations may then be done concurrently and will use a Source ULA from the Clear_To_Send Operation as the Destination ULA. Using this substitution method in combination with a dedicated control channel may also prevent or reduce blocking effects where the underlying physical medium suffers from high latency.

B.4 Many-to-many striping

Figure B.3 shows many-to-many striping as the combination of the one-to-many and many-to-one striping. The system receiving data indicates its desire to receive in a striped fashion by issuing multiple Clear_To_Send Operations with differing return interface addresses. The system sending data chooses to stripe by sending from multiple interfaces that are capable of reaching the proper destination.

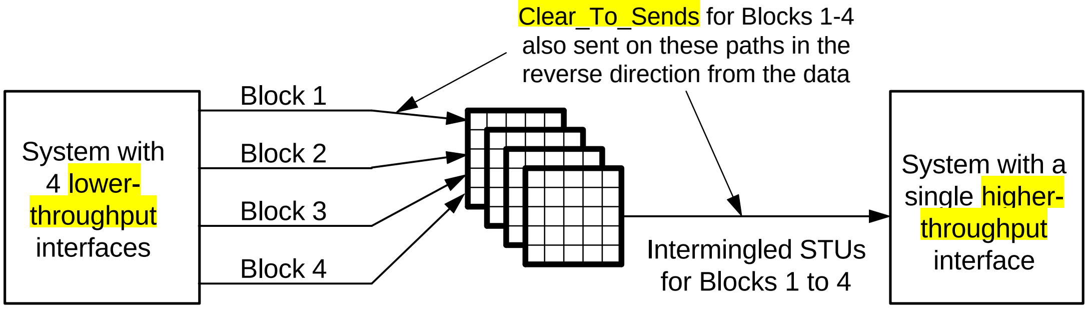
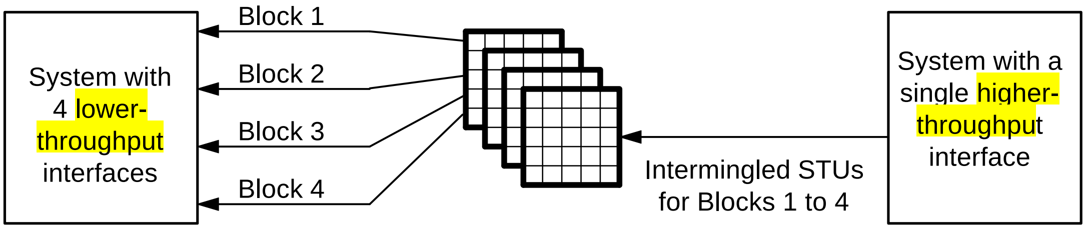
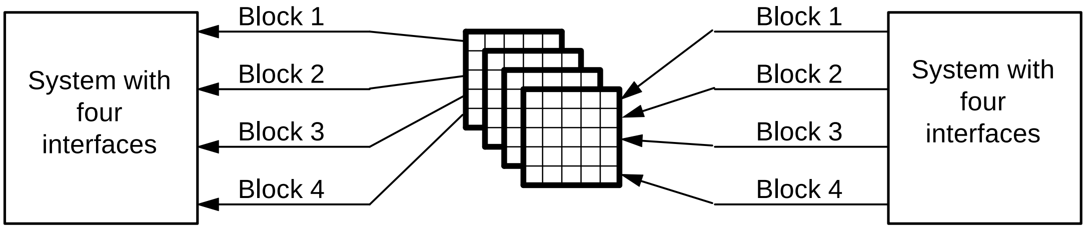


Figure B.1 – Many-to-one striping



Clear_To_Send for each Block is sent in the reverse direction on the same path each Block traverses (or is made to appear that way)

Figure B.2 – One-to-many striping



Clear_To_Send for each Block is sent in the reverse direction on the same path each Block traverses (or is made to appear that way)

Figure B.3 – Many-to-many striping

Annex C
(informative)

Scheduled Transfer example

We were not able to complete all of the edits on Annex C, but it is a start. No margin bars or highlights.

The following examples demonstrate various aspects of the scheduled transfer mechanism. The examples are intended for generic application, but may contain some implementation specific ideas. The examples included are:

– Detailed simple transfer example (see C.1)

– Persistent memory example (see C.2)

– Striped Ethernet to HIPPI-6400 example (see C.3)

– Striped HIPPI-800 to HIPPI-6400 example (see C.4)

C.1 Detailed simple transfer example

This example demonstrates the basic Scheduled Transfer Operations. Device X is sending a 256 KB file to device Y.

– Fields that contain a * are unused and transmitted as zeroes.

– A hex value that takes up fewer bits than its field has zeroes in the upper bits.

– Table C.1 shows all fields and corresponding values for the numbered Operations shown below. The Operation description and table location are correlated by the numbers in parentheses.

C.1.1 Virtual Connection set up

Device Y initiates a Virtual Connection setup with device X using a Request_Port Operation. The Virtual Connection setup is a dual direction connection and either device can begin a Scheduled Transfer after setup, independent of which device initiated the Virtual Connection.

(1) Y->X

Request_Port (
Op: ‘x01’,
A-Slots: SlotsY,
B-Port: x’0000’,
A-Port: PortY,
A-Bufsize: x’10’ *(BufsizeY = 64 KB)*,
A-Key: KeyY,
A-Max-STU: x’10’ *(Max-STUY = 64 KB)*,
EtherType: B_num)

Device Y provides its Key, Port, available slots, Bufsize, Max-STU, and the EtherType for this Virtual Connection to device X. X binds to the well-known Port x'0000' which is used for Port setup. Based on the EtherType a corresponding Port value is mapped and sent in the Request_Port_Response. Device X responds to the request with a Request_Port_Response.

(2) X->Y

Request_Port_Response (
Op: x‘02’
B-Slots: SlotsX
A-Port: PortY,
B-Port: PortX,
A-Key: KeyY,
B-Bufsize: x’0E’ *(BufsizeX = 16 KB)*,
B-Key: KeyX,
B-Max-STU: x’0E’ *(Max-STUX = 16 KB)*)

Device X provides its Key, Port, available Slots, Bufsize, and Max-STU to device Y. Y binds to KeyY and PortY to associate the response with the above Request_Port (as opposed to other Request_Ports that Y may have initiated).

A dual direction Virtual Connection has been established and both sides know the other's Key, Port, available Slots, Bufsize, and Max-STU. The Keys are an authentication value which will stop invalid Operations, (e.g., an inadvertent Port_Teardown Operation might destroy the

Virtual Connection and all ongoing transfers). The Port values are used to map to upper-layer entities and may be the same mapping used for Internet style Ports. The Slot value gives the destination flow control power over all Operations requiring a Slot. The Max-STU provides a means for device X and device Y to declare the maximum size of data payloads it can accept, independent of its buffer size. The Bufsize parameter (buffer size) provides buffer size information needed for. The intermediate device has a smaller STU size.

Figure C.1 summarizes the information provided by each device during the Virtual Connection set up.

Virtual Connection values

Device X	Device Y	Function
SlotsX	SlotsY	Slot flow control
PortX	PortY	Port binding
KeyX	KeyY	Authentication
BufsizeX	BufsizeY	Buffer size
Max-STUX	Max-STUY	STU size limit

Figure C.1 – Virtual Connection information exchanged

C.1.2 Scheduled Transfer set up

Device X, the Originating Source, initiates a 256 KB Transfer using the Virtual Connection established above to device Y, the Final Destination.

(3) X->Y

Request_To_Send (
Op: x’16’,
Flags: D2 *(Data channel = 2)*,
R_Port: PortY,
S_Port: PortX,
Key: KeyY,
S_id: idX,
T_len: x’40000’ *(256 KB)*)

Device Y expects the data to be delivered on Data Channel 2 as specified in the Flags parameter. Device Y reads the Transfer length (256 KB), agrees to accept the Transfer, and responds with a Request_To_Send_Response.

(4) Y->X

Request_To_Send_Response (
Op: x’17’,
S_count: x’11’ *(128 KB Blocksize)*,
R_Port: PortX,
S_Port: PortY,
Key: KeyX,
R_id: idX,
S_id: idY,

Device Y selects a Blocksize of 2^{17} (128 KB). The 256 KB Transfer will consist of two 128 KB Blocks.   Device Y assigns an identification (idY) which it can use to quickly identify the correct transfer.

All further Scheduled Transfer Operations will continue to use the appropriate binding information: Key, Ports, and the identifications but they are not shown in the remaining steps of this example. See table C.1 for the exhaustive list of parameters for each operation.

C.1.3 Block 0 transfer

Device Y sends a Clear_To_Send Operation once it has finished allocating resources for the Transfer.

(5) Y->X

Clear_To_Send (
Op: x’1A’,
S_count: x’11’ *(128 KB Blocksize)*,
R_Port: PortX,
S_Port: PortY,
Key: KeyX,
XBufx: Bufval,
Offset: x’0’,
B_num: x’0’,
OS_Offset: x’0’)

The Clear_To_Send contains the Final Destination's (device Y) starting buffer index (Bufx), the Initial Offset, and the Offset for this Block.

In this example, an efficient tiling between the two nodes is apparent. Device X can send eight 16 KB STUs to fill a Block in device Y. However, HIPPI-ST accommodates a large range of implementations by allowing any number and size of STUs to fill a Block with the following requirements: an STU may not exceed the Max-STU size, 64 KB in this case; and an STU may not be sent to the Final Destination that would overrun a Buffer boundary, Block boundary, or Transfer boundary. For the first Block, the Originating Source, limited by its own buffer size, sends eight 16 KB STUs.

(6-13 all Data Operations) X->Y

#	Flags	S_count	Bufx	Offset	B_num
(6)	T,D2	x'0'	Bufval	x'0'	x'0'
(7)	T,D2	x'1’	Bufval	x'4000'	x'0'
(8)	T,D2	x'2'	Bufval	x'8000'	x'0'
(9)	T,D2	x'3'	Bufval	x'C000'	x'0'
(10)	T,D2	x'4'	Bufval+1	x'0'	x'0'
(11)	T,D2	x'5'	Bufval+1	x'4000'	x'0'
(12)	T,D2	x'6'	Bufval+1	x'8000'	x'0'
(13)	L,D2	x'7'	Bufval+1	x'C000'	x'0'

As can be seen in this example, the Originating Source does most of the work to align the Buffer regions in the Destination. Figure C.2 shows how the first Block fits into Y's Bufx regions. The last STU of the Block has Silent = 0 so that a Slot resource will be allocated for this STU (informing the ULP of Block reception).

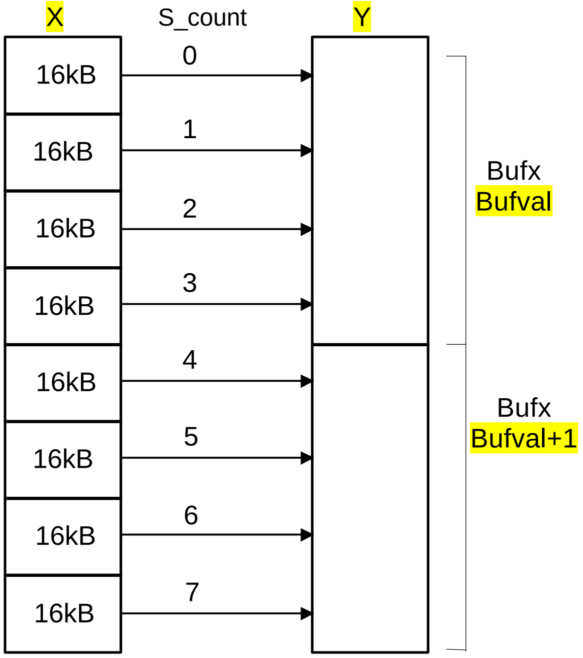


Figure C.2 – Block 0 buffer tiling

C.1.4 Block 1 Clear_To_Send

The second Clear_To_Send allows device X to begin sending the second Block. Because the two Clear_To_Sends contain no overlapping buffer regions (in this example), they could have been issued one after another. The second Clear_To_Send could also use the same Bufx as the first one, but only if it waits for the first Block to complete before issuing the second Clear_To_Send.

Open Issue - Should the second Block start at a completely different Bufx range, e.g., Bufval+50, to reinforce the idea of each Block being indepent of each other?

(14) Y->X

```
Clear_To_Send (
     Op: x’1A’,
     S_count: x’11’ (128 KB Blocksize),
     R_Port: PortX,
     S_Port: PortY,
     Key: KeyX,
     Bufx: Bufval2,
     Offset: x’0’,
     Sync: B_id
     B_num: x’1’,
     OS_Offset: x’0’)
```

For the second Block, the Originating Source grows a buffer gather mechanism which can pull two buffers at a time. The resulting Data operations appear below.

(15-18 are all Data Operations) X->Y

#	Flags	S_count	Bufx	Offset	B_num
(15)	T,D2	x'0'	Bufval2	x'0'	x'1'
(16)	T,D2	x'1'	Bufval2	x'8000'	x'1'
(17)	T,D2	x'2'	Bufval2+1	x'0'	x'1'
(18)	S,L,D2	x'3'	Bufval2+1	x'8000'	x'1'

Figure C.3 shows the resulting tiling.

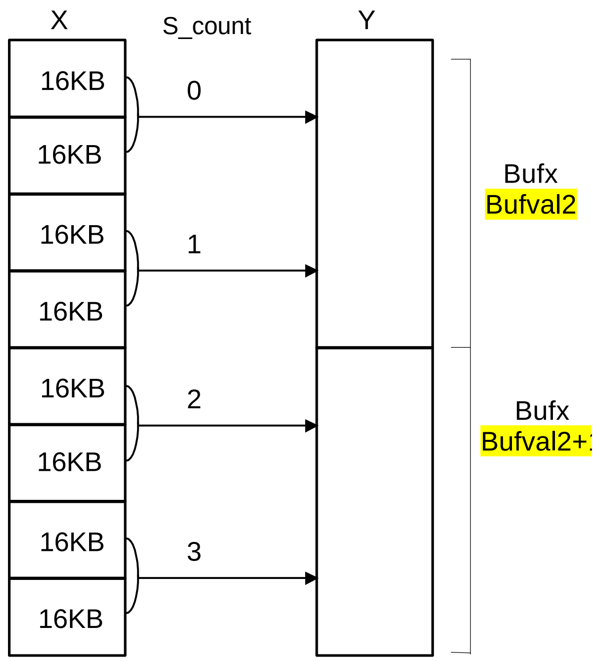


Figure C.3 – Block 1 buffer tiling

The last Data Operation contains the Send_State flag which triggers device Y to update the Slot value and acknowledge the last Block received. Each of the Data Operations carries a synchronization value in the Sync parameter which the Request_State_Response will echo so device X can synchronize its Slot parameter with the number of outstanding Operations.

(19) Y->X

Request_State_Response (
Op: x’1D’,
S_count: C-SlotsY,
R_Port: PortX,
S_Port: PortY,
Key: KeyX,
Offset: x’1’,
Sync: SyncX,
B_num: x’1’)

The Request_State_Response contains an updated Slot value, called C-SlotsY. The Block number acknowledges this Block and all lower numbered Blocks. The B_seq contains the highest received Block number in a contiguous set from the first Block. The value in B_num acknowledges the Block that had the Send_State flag set.

C.1.5 Ending the Virtual Connection

Either side can terminate the Virtual Connection and free all of the resources associated with the Virtual Connection.

(20) Y->X

Port_Teardown (
Op: x’03’,
R_Port: A-Port,
S_Port: B-Port,
Key: A-Key)

(21) X->Y

Port_Teardown_ACK (
Op: x’04’,
R_Port: B-Port,
S_Port: A-Port,
Key: B-Key)

(22)Y->X

Port_Teardown_Complete (
Op: x’05’,
R_Port: A-Port,
S_Port: B-Port,
Key: A-Key)

Table C.1 – Scheduled Transfer example summary

Operation	Op	Flags	S_count	R_Port	S_Port	Key	R_id	S_id	Bufx	Offset	Sync	B_num
(1) Request_Port (Y->X)	***x’01’***	*	***SlotsY***	***x’0000’***	***PortY***	***	***	***	***x’10’*** ***(64 KB)***	***KeyY***	***x’10’*** ***(64 KB)***	***Ether-Type***
(2) RQP_Response (X->Y)	x’02’	*	SlotsX	PortY	PortX	KeyY	*	*	x’0E’ (16 KB)	KeyX	x’0E’ (16 KB)	*
(3) Request_To_Send (X->Y)	x’16’	D2	*	PortY	PortX	KeyY	*	idX	*	*	x’40000’	
(4) RTS_Response (Y->X)	***x’17’***	***	***x’11’***	***PortX***	***PortY***	***KeyX***	***idX***	***idY***	***	***	***	***
(5) Clear_To_Send (Y->X)	***x’1A’***	***	***x’11’***	***PortX***	***PortY***	***KeyX***	***idX***	***idY***	***Bufval***	***x’0’***	***	***x’0’***
(6) Data (X->Y)	x'1B'	T, D2	0	PortY	PortX	KeyY	B_id1	idX	Bufval	x’0’	Syncval	x’0’
(7) Data (X->Y)	x'1B'	T, D2	1	PortY	PortX	KeyY	B_id1	idX	Bufval	x’4000’	Syncval	x’0’
(8) Data (X->Y)	x'1B'	T, D2	2	PortY	PortX	KeyY	B_id1	idX	Bufval	x’8000’	Syncval	x’0’
(9) Data (X->Y)	x'1B'	T, D2	3	PortY	PortX	KeyY	B_id1	idX	Bufval	x’C000’	Syncval	x’0’
(10) Data (X->Y)	x'1B'	T, D2	4	PortY	PortX	KeyY	B_id1	idX	Bufval+1	x’0000’	Syncval	x’0’
(11) Data (X->Y)	x'1B'	T, D2	5	PortY	PortX	KeyY	B_id1	idX	Bufval+1	x’4000’	Syncval	x’0’
(12) Data (X->Y)	x'1B'	T, D2	6	PortY	PortX	KeyY	B_id1	idX	Bufval+1	x’8000’	Syncval	x’0’
(13) Data (X->Y)	x'1B'	L, D2	7	PortY	PortX	KeyY	B_id1	idX	Bufval+1	x’C000’	Syncval	x’0’
(14) Clear_To_Send (Y->X)	***x’1A’***	***	***x’11’***	***PortX***	***PortY***	***KeyX***	***idX***	***idY2***	***Bufval+2***	***x’0’***	***	***x’1’***
(15) Data (X->Y)	x'1B'	T, D2	0	PortY	PortX	KeyY	B_id2	idX	Bufval2	x’0’	Syncval	x’1’
(16) Data (X->Y)	x'1B'	T, D2	1	PortY	PortX	KeyY	B_id2	idX	Bufval2	x’8000’	Syncval	x’1’
(17) Data (X->Y)	x'1B'	T, D2	2	PortY	PortX	KeyY	B_id2	idX	Bufval2+1	x’0’	Syncval	x’1’
(18) Data (X->Y)	x'1B'	S,L, D2	3	PortY	PortX	KeyY	B_id2	idX	Bufval2+1	x’8000’	SyncX	x’1’
(19) Request_State_ Response (Y->X)	***x'1D'***	***	***C-SlotsY***	***PortX***	***PortY***	***KeyX***	***idX***	***idY2***	***	***x’1’***	***SyncX***	***x’1’***
(20) Port_Teardown (Y>X)	***x’03’***	***	***	***PortX***	***PortY***	***KeyX***	***	***	***	***	***	***
(21) Port_Teardown_ACK (X->Y)	x’04’	*	*	PortY	PortX	KeyY	*	*	*	*	*	*
(22) Port_Teardown_ Complete (Y->X)	***x’05’***	***	***	***PortX***	***PortY***	***KeyX***	***	***	***	***	***	***

NOTE – Operations from X to Y are shown in plain text; Operations from Y to X are shown in bold italic.

The Port_Teardown is a three-way handshake that decreases timeout dependency for releasing resources. The device sending the Port_Teardown_ACK can release all of the resources upon reception of the Port_Teardown_Complete.

C.2 Persistent memory example

Though not detailing a shared memory structure, the following example presents the setup of a buffer region that handles both reads and writes, a building block for shared memory operations. Figure C.4 shows the devices and Bufx range used in this example.

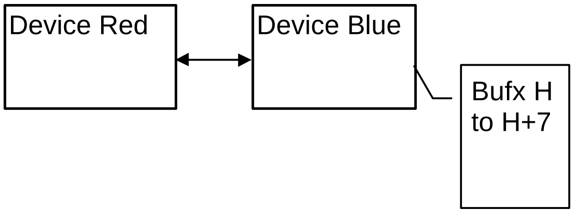


Figure C.4 – Persistent memory setup

C.2.1 Set up

The following commands set up the Virtual Connection:

(1) Red → Blue

Request_Port (
Op: x'01',
Slots: SlotsRed,
R_Port: Port,
S_Port: Port,
Bufsize: x'0E' *(16 KB)*,
XKey: KeyRed,
Max-STU: x'0E' *(16 KB)*,
Ethertype: Local)

(2) Blue → Red

Request_Port_Response (
Op: x'02',
Flags: P *(Persistent)*,
Slots: SlotsBlue,
R_Port: Port,
S_Port: Port,
Key: KeyRed,
Bufsize: x'0D' *(8 KB)*,
XKey: KeyBlue,
Max-STU: x'0D' *(8 KB)*)

Device Blue flags that it accepts Persistent Request_To_Send transfers. The rest of the parameters are summarized in table C.2

Table C.2 – Red / Blue Virtual Connection parameters

Parameter	Red	Blue
Slots	SlotsRed	SlotsBlue
Ports	Port	Port
Keys	KeyRed	KeyBlue
Bufsize	16 KB	8 KB
Max-STU	16 KB	8 KB

Device Red would like to read and write 64 KB memory segments on device Blue and issues a Request_To_Send.

(3) Red → Blue

Request_To_Send (
Op: x'16',
Flags: P,D1 *(Persistent, Channel 1)*,
S_id: idJ,
T_len: x'10000' *(64 KB)*)

Device Blue maps the memory, returns a Bufx value pointing to the first and returns the Bufx to Red in a Clear_To_Send Operation.

(4) Blue → Red

Clear_To_Send (
Op: x'1A',
Blocksize: x'10' *(64 KB)*,
R_Port: Port,
S_Port: Port,
Key: KeyRed,
R_id: idJ,
S_id: idtH,
Bufx: BufxH,
Offset: x'0',
B_id: x'10000' *(64 KB)*,
B_num: x'0',
I_Offset: x'0')

Device Blue has pinned down the persistent memory allowing device Red to either read (with an Request_To_Receive Operation) or write (with a Data Operation). Note that with Persistent = 1, Data Operations can be issued without being preceded by a Clear_To_Send Operation from the other end.

C.2.2 Reading

Device Red would first like to read the established persistent memory Block prior to changing memory contents and initiates a Request_To_Receive to device Blue.

(5) Red → Blue

Request_To_Receive (
Op: x’18’,
Flags: P,D1 *(Channel 1)*,
R_Port: Port,
S_Port: Port,
Key: KeyBlue,
R_id: bidH,
S_id: idJ,
Bufx: BufxH,
Offset: x’0’,
B_id:
Sync: x’10000’ *(64 KB)*,
OS_Bufx: BufxJ,
OS_Offset: x’0’)

Blue receives the Request_To_Receive Operation and responds with a number of Data Operations. Each STU is 8 KB, limited by Blue’s send buffer mechanism. Hence, in this example eight Data Operations will be used, i.e., 64 KB Block size divided by the STU size. Only the first Data Operation is shown in detail, parameter differences in the following ones are listed below it.

(6) Blue → Red

Data (
Op: x’1B’,
Flags: T,D1 (*Silent, Channel 1*)
S_count: x’0’,
R_Port: Port,
S_Port: Port,
Key: KeyRed,
R_id: bidH,
S_id: idH,
Bufx: BufxJ,
Offset: x’0’,
B_num: x’0’)

#	Flags	S_count	Bufx	Offset
7	T,D1	x'1'	BufxJ	x'2000'
8	T,D1	x'2'	BufxJ+1	x'0'
9	T,D1	x'3'	BufxJ+1	x'2000'
10	T,D1	x'4'	BufxJ+2	x'0'
11	T,D1	x'5'	BufxJ+2	x'2000'
12	T,D1	x'6'	BufxJ+3	x'0'
13	T,L,D1	x'7'	BufxJ+3	x'2000'

Device Red has successfully read the contents of the memory location set up on device Blue. Blue could have set the Send_State flag in the last Data Operation if it cared to receive an acknowledgment that device Red completed the read.

C.2.3 Writing

Next, Red would like to change the memory contents on Blue and sends a Data Operation without Block flow control or other setup (after the initial Request_To_Send, Clear_To_Exchange). Each STU is 8 KB to fill one of device Blue’s buffers. Only the first Data Operation is shown in detail, parameter differences in the ones following are listed below it.

(14) Red → Blue

Data (
Op: x’1B’,
Flags: T,D1,
S_count: x’0’,
R_Port: Port,
S_Port: Port,
Key: KeyBlue,
R_id: idH,
S_id: idJ,
Bufx: BufxH,
Offset: x’0’,
B_num: x’0’)

#	Flags	S_count	Bufx	Offset
15	T,D1	x'1'	BufxH	x'2000'
16	T,D1	x'2'	BufxH+1	x'0'
17	T,D1	x'3'	BufxH+1	x'2000'
18	T,D1	x'4'	BufxH+2	x'0'
19	T,D1	x'5'	BufxH+2	x'2000'
20	T,D1	x'6'	BufxH+3	x'0'
21	S,L,D1	x'7'	BufxH+3	x'2000'

The last Data Operation sets the Send_State flag to confirm reception of the write. Silent = 0 is used to alert device Blue that the location has been written.

C.2.4 Closing the persistent memory

Either side can close the persistent memory by issuing an END operation. Port_Teardown Operations will also close the persistent memory.

C.3 Translated, striped Ethernet to HIPPI-6400 example

This example shows multiple Gigabit Ethernet links and aggregated them onto HIPPI-6400. The actions of both end devices and the translator are detailed in this example. Figure C.5 shows the network topology for this example.

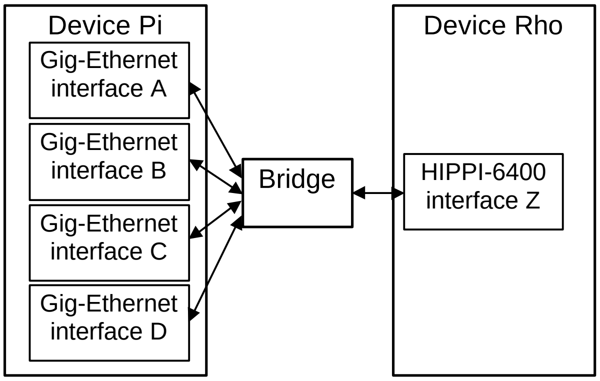


Figure C.5 – Network topology

Pi and Rho are two end devices that need to communicate large amounts of data. Unfortunately, Pi only speaks Gigabit Ethernet and Rho only knows HIPPI-6400. A transparent, Gigabit-Ethernet-switching translator is placed between them to break the MAC protocol barrier. To enable high throughput direct memory access (DMA) transfers, the devices use the HIPPI-ST protocol.

The translator will take the HIPPI-6400 Request_Port Operation, look up the correct Gigabit Ethernet port for A, reformat the HIPPI-6400 message as a Gigabit Ethernet packet, and forward the packet to Gigabit Ethernet interface A.

The following example demonstrates data transfers moving in both directions between Pi and Rho.

C.3.1 Virtual Connection Setup

Rho initiates the Virtual Connection setup to Pi's interface A ULA address (found using ARP for Pi). Interface A will act as the control channel interface as well as one of the Data pipes. A non-Data control interface could also be used to reduce any contention between Control and Data operations on interface A.

Rho inserts a zero into the Max-Blocksize (carried in the OS_Bufx) parameter. The translator checks its available buffers for this port and inserts a Max-Blocksize value of x'13' (512 KB) into the Request_Port before forwarding it to Pi.

```
Z → A

Request_Port (
      Op: x'01',
      Flags: O (Out_of_Order),
      Slots: SlotsRho,
      R_Port: x'0000',
      S_Port: PortRho,
      Bufsize: x'0E' (16 KB),
      XKey: KeyRho,
      Max-STU: x'0E' (16 KB),
      Ethertype: Ethertype,
      Max-Blocksize: x'0' → x'13' (512 KB))
```

Open Issue – Aghh! What's this new parameter? The Max-Blocksize states the maximum data that can be stored and forwarded through the translator without a strict flow control mechanism running between the translator and both ends. This means that the translator can have less complicated ST protocol hardware, but requires large, high-speed buffers.

Pi processes the request and sends a Request_Port_Response from interface A. The translator receives the Request_Port_Response, converts the Gigabit Ethernet packet format to a HIPPI-6400 message, and transmits the message on Virtual Channel 0 (because it's a Control Operation). Again, the translator overwrites the Max-Blocksize value.

A → Z

```
Request_Port_Response (
      Op: x’02’,
      Flags: O,
      Slots: SlotsPi,
      R_Port: PortRho,
      S_Port: PortPi,
      Key: KeyRho,
      Bufsize: x’0C’ (4 KB),
      XKey: KeyPi,
      Max-STU: x’0A’ (1 KB),
      Max-Blocksize: x’0’ → x’13’ (512 KB))
```

Pi has a 4 KB Bufsize and selects a 1 KB Max-STU size to accommodate the 1500 byte size constraint of Gigabit Ethernet packets. The Virtual Connection is established and both devices have indicated the ability to receive Blocks out of order (required for translated striping). Table C.3 summarizes the parameters exchanged.

Table C.3 – Pi / Rho Virtual Connection parameters

Parameter	Pi	Rho
Slots	SlotsPi	SlotsRho
Ports	PortPi	PortRho
Keys	KeyPi	KeyRho
Bufsize	4 KB	16 KB
Max-STU	1 KB	16 KB
Max-Blocksize	512 KB	512 KB

C.3.2 Sending to Rho

Pi needs to send a one megabyte Transfer to Rho.

A → Z

```
Request_To_Send (
      Op: x’ 16’,
      Flags: D1,
      Outstanding_Blocks: x’10’,
      S_id: idt1C,
      T_len: x’100000’ (1 MB))
```

Open Issue – Note the new parameter in S_count called “Outstanding Blocks”. The Orig. Source requires a means to “advise” the Final Destination as to Blocksize selection to keep the Orig. Source’s interfaces busy.

Pi selects to send Data Operations through Virtual Channel 1. Pi sets his preferred number of outstanding blocks to 16, hopefully, keeping four blocks outstanding for each of the four interfaces. The Outstanding Blocks parameter is only a suggestion and may be ignored by Rho.

A total of 16 Clear_To_Send Operations are sent from Z to A. Only the first Clear_To_Send is shown in detail, parameter differences in the ones following are listed below it.

Z → A

```
#1, Clear_To_Send (
      Op: x’1A’,
        Blocksize: x’0F’ (32 KB),
        R_id: idt1C,
        S_id: idt1H,
        Bufx: Rho0,
        Offset: x’0’,
        B_num: x’0’)
```

#	Bufx	B_num
2	Rho2	x'1'
3	Rho4	x'2'
4	Rho6	x'3'
5	Rho8	x'4'
6	Rho10	x'5'
7	Rho12	x'6'
8	Rho14	x'7'
9	Rho16	x'8'
10	Rho18	x'9'
11	Rho20	x'A'
12	Rho22	x'B'
13	Rho24	x'C'
14	Rho26	x'D'
15	Rho28	x'E'
16	Rho30	x'F'

Rho selects a Blocksize of 32 KB based on his buffer size, the Max-Blocksize, and his ability to handle interrupts. Rho sends 16 Clear_To_Sends to interface A meeting the requested number of outstanding Blocks, while staying under the Max-Blocksize.

Pi begins receiving the Clear_To_Send requests through interface A and in a round-robin fashion farms Data Operation requests to each of the four Gigabit Ethernet interfaces. Only the first Data Operation is shown in detail, parameter differences in the ones following are listed below it.

From here on out, the example needs to be compressed. Many of the following commands share the same parameters and will be made into a table listing only the differences to make it easier to read.

A → Z

Data (
Flags: T,D1,
S_count: x'0',
R_id: idt1H,
S_id: idt1C,
Bufx: Rho0,
Offset: x'0',
B_num: x'0')

B → Z

Data (
Flags: T,D1,
S_count: x'0',
R_id: idt1H,
S_id: idt1C,
Bufx: Rho2,
Offset: x'0',
B_num: x'1')

C → Z

Data (
Flags: T,D1,
S_count: x'0',
R_id: idt1H,
S_id: idt1C,
Bufx: Rho4,
Offset: x'0',
B_num: x'2')

D → Z

Data (
Flags: T,D1,
S_count: x'0',
R_id: idt1H,
S_id: idt1C,
Bufx: Rho6,
Offset: x'0',
B_num: x'3')

A → Z

Data (
Flags: T,D1,
S_count: x'1',
R_id: idt1H,
S_id: idt1C,
Bufx: Rho0,
Offset: x'400' *(1 KB)*,
B_num: x'0')

… (506 1 KB STU's later) …

D → Z

Data (
Flags: S,L,D1,
S_count: x'31',
R_id: idt1H,
S_id: idt1C,
Bufx: Rho31,
Offset: x'3C00' *(15 KB)*,
B_num: x'15')

Each of the Data Operations will contain 1 KB of STU for the simplest tiling scheme (due to the Gigabit Ethernet packet size restriction). The flag bits carry the data channel value so the translator can correctly place each STU on the correct Virtual Channel. All STU's are marked silent except for the last of each Block which requests a Request_State_Response acknowledgment. The STU count will range from 0 to 31 for each block (32 each 1 KB STU's). The last Data operation listed above shows interface D sending the last STU of Block 15 (halfway through the transfer). The order that the Blocks are sent may not match the order shown above which is why Rho must accept out of order Blocks. The translator is required to switch between the incoming Gigabit Ethernet interfaces onto VC1 of the HIPPI-6400 interface. The aggregate bandwidth of the striped Gigabit Ethernets sits around half the bandwidth of the HIPPI-6400 link and things should flow smoothly, (note the translator's capability to buffer all the outstanding blocks in case a lack of credits appears on the HIPPI-6400 link).

As groups of 32 1 KB STU's fill in each Block in the HIPPI-6400 receive buffers, new blocks may be cleared for transmission (as long as the 512 KB translator buffer is not overrun). Blocks in error may be resent according to normal HIPPI-6400 retransmission methods.

Rho will clear another 16 Blocks for transmission to complete the megabyte Transfer. Acknowledgments are sent in the normal manner.

C.3.3 Sending to Pi

Rho received the first transfer and after changing a bit, sends the megabyte of data back to Pi.

Z → A

Request_To_Send (
Flags: D2,
Outstanding_Blocks: x'0',
S_id: idt2H,
T_len: x'100000' *(1 MB)*)

Rho plans to send on Data Channel 2 and sets the outstanding Blocks to zero showing that Rho doesn't care. This will allow Pi to select a Blocksize that is not constrained by a set number of Blocks, (but still by the Max-Blocksize).

A → Z

Clear_To_Send (
Blocksize: x'11' *(128 KB)*,
R_id: idt2H,
S_id: idt2C,
Bufx: Pi0,
Offset: x'0',
B_num: x'0')

B → Z

Clear_To_Send (
Blocksize: x'11' *(128 KB)*,
R_id: idt2H,
S_id: idt2C,
Bufx: Pi32,
Offset: x'0',
B_num: x'1')

C → Z

Clear_To_Send (
Blocksize: x'11' *(128 KB)*,
R_id: idt2H,
S_id: idt2C,
Bufx: Pi64,
Offset: x'0',
B_num: x'2')

D → Z

Clear_To_Send (
Blocksize: x'11' *(128 KB)*,
R_id: idt2H,
S_id: idt2C,
Bufx: Pi96,
Offset: x'0',
B_num: x'3')

A → Z

Clear_To_Send (
Blocksize: x'11' *(128 KB)*,
R_id: idt2H,
S_id: idt2C,
Bufx: Pi128,
Offset: x'0',
B_num: x'4')

B → Z

Clear_To_Send (
Blocksize: x'11' *(128 KB)*,
R_id: idt2H,
S_id: idt2C,
Bufx: Pi160,
Offset: x'0',
B_num: x'5')

C → Z

Clear_To_Send (
Blocksize: x'11' *(128 KB)*,
R_id: idt2H,
S_id: idt2C,
Bufx: Pi192,
Offset: x'0',
B_num: x'6')

D → Z

Clear_To_Send (
Blocksize: x'11' *(128 KB)*,
R_id: idt2H,
S_id: idt2C,
Bufx: Pi224,
Offset: x'0',
B_num: x'7')

Each of Pi's interfaces responds with two Clear_To_Send's containing its ULA in the Source Address (for the correct return of Data Operations to each interface). The Blocksize of 128 KB means that only four Blocks may be outstanding without overrunning the translator's Max-Blocksize value. Because eight Blocks have been cleared, the translator will queue the last four Clear_To_Send's until the previous ones have been sent by the translator. Since only a single Block will remain outstanding to any of the Gigabit Ethernet interfaces, some lag time will result in clearing the next Block.

```
Z → A

Data(
    Flags: T,D2,
    S_count: x’0’,
    R_id: idt2C,
    S_id: idt2H,
    Bufx: Rho0,
    Offset: x’0’,
    B_num: x’0’)

Z → B

Data(
    Flags: T,D2,
    S_count: x’0’,
    R_id: idt2C,
    S_id: idt2H,
    Bufx: Rho2,
    Offset: x’0’,
    B_num: x’1’)

Z → C

Data(
    Flags: T,D2,
    S_count: x’0’,
    R_id: idt2C,
    S_id: idt2H,
    Bufx: Rho4,
    Offset: x’0’,
    B_num: x’2’)

Z → D

Data(
    Flags: T,D2,
    S_count: x’0’,
    R_id: idt2C,
    S_id: idt2H,
    Bufx: Rho6,
    Offset: x’0’,
    B_num: x’3’)

Z → A

Data(
    Flags: T,D2,
    S_count: x’1’,
    R_id: idt2C,
    S_id: idt2H,
    Bufx: Rho0,
    Offset: x’400’ (1 KB),
    B_num: x’0’)
…(506 1 KB STU’s later) …

Z → D

Data(
    Flags: S,L,D2,
    S_count: x’127’,
    R_id: idt2C,
    S_id: idt2H,
    Bufx: Rho31,
    Offset: x’1FC00’ (127 KB),
    B_num: x’3’)
```

Obeying the Max-STU, Rho sends 1024 1 KB STU’s, but the larger blocks may cause lulls during the transfer when Rho has not received a new Clear_To_Send to keep things going.

NOTES –

1 – In order for people to actually implement and make things useful, striping needs to be kept simple or it will be the option that doesn’t happen. I believe this partially includes somewhat restricting the “applicable in all situations” attributes that are nice, but not always necessary.

2 – Setting Clear_To_Send's small enough to keep Blocks in motion without overrunning the Max-Blocksize parameter.

3 – Efficient use of the translator’s buffers must be maintained. The Max-Blocksize communicated during port setup must be shared by all transfers. The value is called Max-Blocksize and not intermediate storage size as the host may produce more Clear_To_Send's worth in data than the translator can buffer (as long as the Blocksize is not larger than the translator’s declared maximum). The translator will queue the Clear_To_Send requests forwarding only the number that can be safely handled by the translator. Once the block for an outstanding Clear_To_Send is sent, the next Clear_To_Send in the queue may proceed to the end host.

4 – This is all new and the only way the author could think of to enable a now semi-transparent translator between ST devices.

5 – A less complex method can be easily applied to simple N to N stripes across the same media. Neither of the suggested new parameters are required for direct N to N striping.